UNIVERSITY OF CALGARY

Formal Models and Implementations of Distributed Shared Memory

by

Steven William Cheng Hum Yuen

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

February, 2009

UNIVERSITY OF
CALGARY

Abstract

Distributed Shared Memory (DSM) is shared memory that is implemented in a decentralized manner. DSM implementations are specified using memory consistency models, a technique that allows us to reason about memory system effects without considering how the memory system is implemented. We introduce a class of memory consistency models, Partition Consistency, that generalizes Pipelined-RAM, Sequential Consistency, and Ahamad's formalization of Processor Consistency. Existing token and priority queue DSM implementations of Sequential Consistency are generalized to implement the Partition Consistency class of models. To prove the correctness of the DSM implementations, we formally model them and introduce new techniques to present and create such proofs. The above formal implementations are used to create direct implementations that are tested on a high performance computing cluster, WestGrid. We test the performance of the protocols using a mutual exclusion algorithm known to be correct on the implemented models. In one case, significant performance gains are obtained by weakening the consistency of an implementation. In the other cases, the implementations fail to improve on the existing Sequential Consistency implementations, but additional insights are gained about the design of DSM implementations.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed shared memory is a type of shared memory. Shared memory can be accessed by more than one process. Distributed shared memory is implemented in a decentralized manner. For example, a shared memory system that consists of a single memory controller chip that accesses a bank of memory chips is very centralized. This thesis presents replicated shared memories implemented on a network of multiprocessor nodes. A replicated shared memory is implemented by storing a complete copy of the shared memory on each node. The replicated shared memory uses a network protocol to the synchronize these copies which we call replicas. Replicated shared memories are decentralized and therefore distributed by nature.

The way that synchronization is managed in a replicated shared memory has serious impacts on the performance and correctness of the programs that use it. This synchronization issue is an instance of a general problem that affects shared memories and concurrent programming in general. The problem is that agreement between processes in a concurrent system has a significant cost in terms of performance. To obtain the best performance, we must find ways to weaken the agreement while maintaining correctness.

Programming for shared memory often requires dealing with levels of complexity that are hidden or don't exist in sequential programming. Three main issues make shared memory programming more difficult: (1) high level sections of processes may execute in arbitrary order, (2) low level operations of processes interfere with each other, and (3) agreement between processes on the order of low level operations may be weak.

High level sections execute in arbitrary order: High level sections of programs are subroutines. They are an important tool for abstraction. If a subroutine is well documented, it may be used without knowing the exact details of how it is programmed.

The first problem when dealing with concurrency is that the assumption of a single process executing the subroutines fails. Consider the following example:

One process decides to add Alice to a list, then print her name. In a sequential system, we can assume that the name at the top of the list is the one that we just inserted.

```
adder()
1   addtofront(list,"Alice")
2   print(first(list))
```

In a concurrent system, another process may add Bob to the list, after Alice is added, but before the first process retrieves the top of the list.

```
addanother()
1   addtofront(list,"Bob")
```

This causes adder() to print "Bob" rather than "Alice" as intended. This error is the symptom of a more general problem, that extends to a lower level of abstraction, machine level operations executed by concurrent processes.

Low level operations of processes may interfere with each other: Low level operations of processes are the individual operations invoked by the processes, the ones managed directly by hardware. The details of these operations are cleanly hidden and abstracted in a sequential system, but are exposed when concurrent processes share memory.

Consider a variable x initially equal to 3. Then introduce two processes that operate on x.

```
multiplybytwo()
1   x ← x × 2
```

multiplybyfour()
1 $x \leftarrow x \times 4$

We expect that after both processes complete $x = 3 \times 2 \times 4 = 24$.

The end result after both processes complete is the same if either of the high level sections execute in either order. Also, each high level section consists of only a single statement, essentially the smallest unit of a program. However, on most processors a statement $x \leftarrow x \times 4$ is implemented as:

compile$(x \leftarrow x \times 4)$
1 $r_1 \leftarrow \texttt{read}(x)$
2 $\texttt{write}(x, r_1 \times 4)$

Where r_1 is a register.

This means it is possible to have the following execution occur:

$$\begin{bmatrix} \text{multiplybytwo}: & \frac{\texttt{read}(x)}{3} & & \texttt{write}(x, 6) \\ \text{multiplybyfour}: & & \frac{\texttt{read}(x)}{3} & \texttt{write}(x, 12) \end{bmatrix}$$

The result will then be either 6 or 12, not the desired result, 24.

Even the level of abstraction offered by a statement in a programming language is broken. Computer science is concerned with abstraction, but being forced to deal with these low level details seems to limit the amount of abstraction possible. This problem could be solved however, with a lock.

lockmultiplybytwo()
1 acquirelock(*lock*)
2 multiplybytwo()
3 releaselock(*lock*)

lockmultiplybyfour()
1 acquirelock(*lock*)
2 multiplybyfour()
3 releaselock(*lock*)

A lock is a mechanism that isolates groups of operations from each other and requires them to execute in sequence. In this case, it ensures that the operations of multiplybytwo() execute separately from multiplybyfour(). For example, when

lockmultiplybytwo acquires the lock, lockmultiplybyfour cannot acquire it until

lockmultipybytwo calls releaselock. Either process could be the first to acquire the lock, but now there are only two possibilities:

$$\begin{bmatrix} \text{lock-m-bytwo :} & \text{a-lock()} & \text{m-bytwo()} & \text{r-lock()} & & & & \\ \text{lock-m-byfour :} & & & & \text{a-lock()} & \text{m-byfour()} & \text{r-lock()} \end{bmatrix}$$

$$\begin{bmatrix} \text{lock-m-bytwo :} & & & & \text{a-lock()} & \text{m-bytwo()} & \text{r-lock()} \\ \text{lock-m-byfour :} & \text{a-lock()} & \text{m-byfour()} & \text{r-lock()} & & & \end{bmatrix}$$

The only non-determinism left is which processor goes first, and this does not affect the end result. We can now regard each subroutine as an isolated piece, that can be specified without its implementation details. This maintains the level of abstraction that we want.

Locks allow us to build abstractions, but they come with their own problems. Systems with more than one lock can easily end up in a deadlock situation. Furthermore, to gain better performance, we may need to vary the pattern of locking. As we make the pattern of locking more intricate, we lose the abstraction that was gained by introducing locks in the first place.

Application, library and compiler writers concerned with high performance must deal with these low level issues. They have to consider the individual operations of a program and a complex issue: how the low level operations are ordered by the memory

system.

Weak agreement on order of low level operations between processes: Different memory systems have many different strategies for managing low level memory operations. A simple memory system will interleave the operations of the processes in the order that they were issued. In many systems however, the situation is more complicated.

Consider a system running two concurrent processes:

Sender

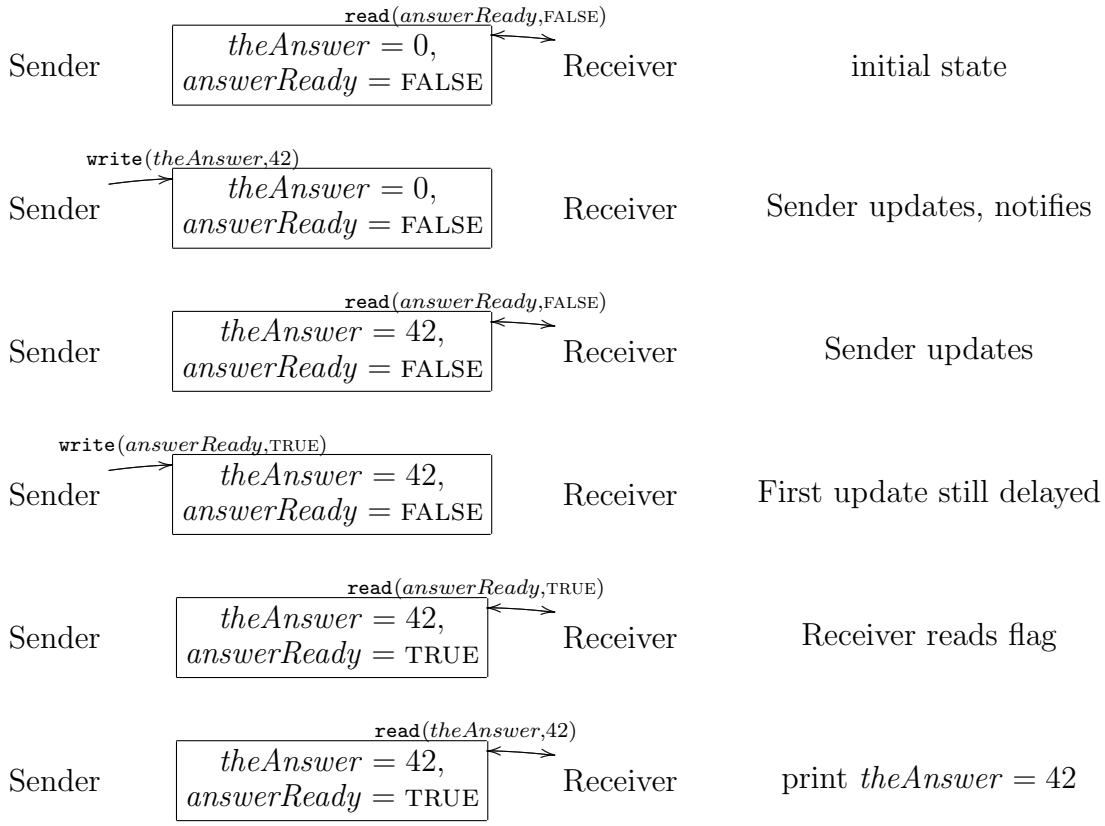
```
1  theAnswer ← 42
2  answerReady ← TRUE
```

Receiver

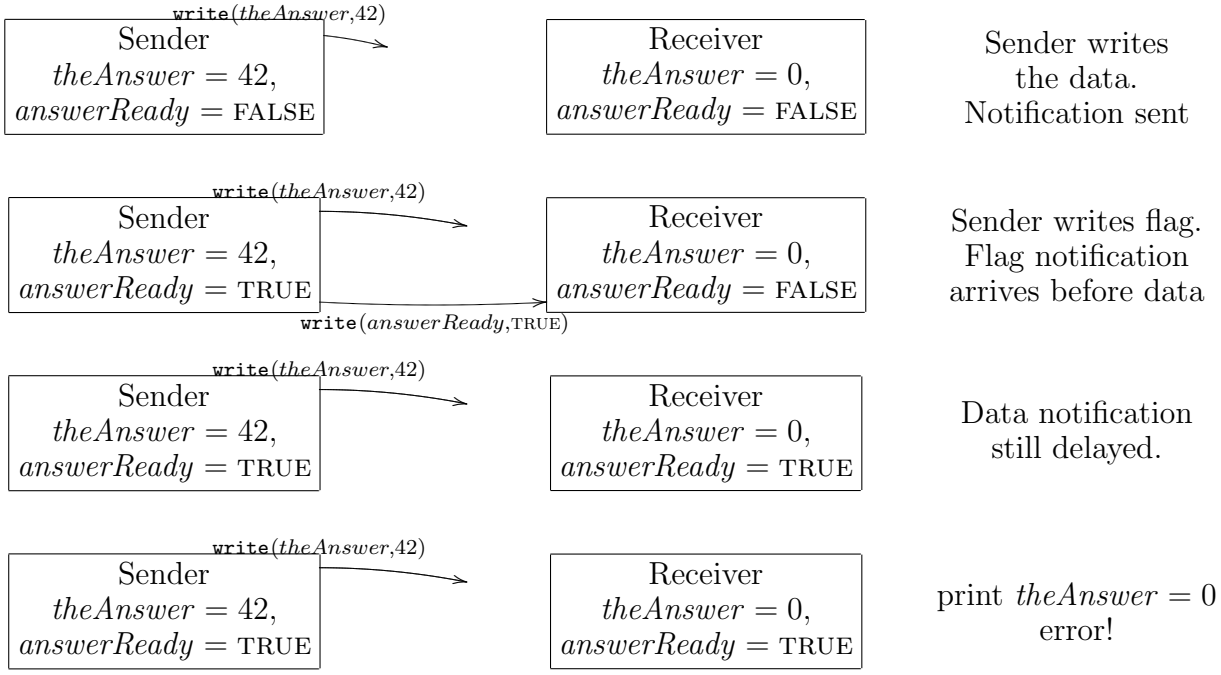
```
1  while ¬ answerReady
2        do skip
3  print(theAnswer)
```

The purpose of this system is that the Receiver process correctly prints out the data that the Sender stored in the *theAnswer* variable.

Consider then a simple memory system that satisfies the programmer's intuition of how shared memory should work. The following is an example execution of the processes running on this system:



Now consider a replicated shared memory with a simple synchronization protocol. Each process runs on a processor with its own local replica of memory. The protocol broadcasts write operations to the other processors when they are performed, and received notifications of write operations are applied when they are received.



This execution ends with an error. It might be said that the memory system is incorrectly implemented, however, strengthening the synchronization protocol adds more overhead. Most replicated shared memory systems do not provide the synchronization seen in the first example, known as sequential consistency [37]. This is also true of distributed shared memories and centralized shared memories in multiprocessor systems.

The result is that in any modern memory system, if we wish to avoid locks, then we are forced to reason about inner workings of the memory system. The type of abstraction that is relied on in sequential programs disappears in this situation. However, there is a way to abstract the relevant issues so that we only deal with the issues that relate to program correctness.

This thesis presents several new Distributed Shared Memory (DSM) implementations. We provide proofs of correctness of these implementations and evaluate their performance compared to existing DSM implementations. To prove the

correctness of the DSM implementations, we formally model them and introduce new techniques to present and create such proofs. The performance evaluation did not provide impressive gains but do provide interesting insights into the nature of DSM implementations.

To prove these implementations correct, we must first formally model them.

1.1 Modeling concurrency and memory consistency models

Memory consistency models allow us to describe the way that a memory system manages low level operations without referring to the system's implementation details. This is done by reasoning about orders on the low level operations. These models also allow us to flexibly build higher level abstractions on these low level operations. Constraints on operation orders are specified without referring to how these orders are maintained.

For example, instead of specifying that a memory system is implemented as a replicated shared memory with FIFO message channels between each processor, the model will only specify that write operations performed sequentially by a process will be seen in the same sequence by all other processors. We call such models *memory consistency models*.

With a memory consistency model we can show that the Sender and Receiver processes described in the previous section are correct without referring to any of the implementation details of the memory system. Memory consistency models allow us to maintain a level of abstraction while dealing with the relevant low level issues.

The details of this modeling will be discussed in Chapter 3. Given this model of systems, we must then have methods of proving their properties. Memory consistency models require different proof techniques which we will introduce.

1.2 Proving correctness of shared memory systems

This thesis discusses general methods of reasoning with memory consistency models applied to specific distributed shared memory implementations. Proving the correctness of a system reduces to proving that it meets a specification. For sequential programs, the specification is an input and an output. Given a specified input, a sequential program provides a specified output.

However, as we have demonstrated, the runtime behaviour of a concurrent system matters as much as its end result. For example, a database server continually receives and responds to queries, rather than processing one specific input and producing one specific output. We instead prove that a concurrent system behaves in a way that is consistent with a specified system.

Our notion of correctness depends on the notion of a program transformation. We implement a specification system on a target system by providing a way to transform programs for the specification system to programs on the target system. This transformation is correct if the behaviour of the target system is consistent with the specification system. This is discussed in detail in Chapter 3.

1.3 Contributions of this thesis

We introduce a class of memory consistency models that includes some standard memory consistency models in the literature as special cases. This class of models is implemented in a few different ways as replicated shared memory.

We present a carefully defined framework for reasoning about memory consistency. The proofs of this system are layered and compose. It is capable of handling several subtleties involved with introducing threads. We introduce a type of diagram that makes the presentation of detailed memory consistency proofs shorter and easier to read, inspired by similar proofs in category theory [11]. The above are applied to the

proof of correctness of four implementations.

We use the above formal implementations to create direct implementations that are tested on a high performance computing cluster, WestGrid. We test the performance of the protocols using a mutual exclusion algorithm known to be correct on the implemented models.

Chapter 2

Related Work

This thesis explores issues in three areas: (1) implementations and performance evaluations of distributed shared memory, (2) modeling concurrent systems using memory consistency models, and (3) proofs of correctness of concurrent systems.

Many implementations of distributed shared memory exist. Memory consistency models are usually used to specify distributed shared memories but proofs that systems satisfy memory consistency models are rare. Proofs of correctness for concurrent systems built using other systems that are specified by memory consistency models are rarer. Many powerful frameworks for proving concurrent systems correct exist in the literature, but most of them require models with strong assumptions about time. We incorporate ideas and proof techniques from the literature to prove the correctness of our distributed shared memory systems implemented using systems that are specified by memory consistency models.

There is a large body of research remotely related to this thesis in these areas. Attiya and Welch provide further background references on proofs of correctness, distributed shared memory implementations, and memory consistency models in their textbook, Distributed Computing [8].

2.1 Implementations of Distributed Shared Memory

Huseynov's Distributed Shared Memory (DSM) webpage [35] tracks the many academic and commercial implementations available.

Attiya and Welch examined the difference between DSM implementations of sequential consistency and linearizability [7]. Sequential consistency and linearizability

are memory consistency models that we describe further in Section 2.2.2. Using message delay as a measure of complexity, lower bounds on the guaranteed cost of various combinations of operations in a replicated shared memory are proved. Operations that do not require message delay are called *fast* operations, other operations are *slow*. A lower bound from Lipton and Sandberg [40] on sequential consistency is presented, with a new lower bound on the cost of operations in a linearizable memory. Both linearizable and sequentially consistent memories must incur some message delay given certain combinations of operations. Attiya and Welch provide an implementation of sequential consistency that is faster than the lower bound shown for linearizable memories, separating the two models.

Lipton and Sandberg [40] introduced one of the first weak memory models with their Pipelined-RAM distributed shared memory implementation. All of the operations of the Pipelined-RAM implementation complete without message delay.

Cholvi, Fernandez, Jimenez, and Raynal present a sequentially consistent DSM protocol [14] that can perform fast writes. The lower bound proved by Attiya and Welch shows that some combinations of operations must require some message delay so it is not possible for reads to be fast if writes are fast. However, Cholvi *et. al.* observe that in some situations, a read can be guaranteed not to incur any message delay, improving the best case performance. The implementation presented by Cholvi *et. al.* stores the entire shared memory in a single token that is circulated among the processors. Each processor has its own copy of memory, which it synchronizes with the token each time it receives it, before passing it on. Writes that have not yet been applied to the token are tracked, but complete immediately, so they are fast. Reads may have to wait for the token to arrive in order to complete, but sometimes the implementation can determine that the read can complete immediately.

Whether or not a read completes without a network delay is determined by the pattern of reads and writes performed by the system. Since there is only one token,

many of the complexities and overheads involved with message passing are avoided. However, sending the entire memory in the token may not be practical for all applications.

Shao, Pierce and Welch use quorums to create a protocol implementing weak registers [48]. They use a layered structure to prove their result. Weak registers do provide some consistency constraints, however, they also use a strong notion of global time (see Section 2.2.2).

Agrawal, Choy, Leong and Singh created the Maya DSM [3] to experiment with weak memory consistency models. They implement several models with Maya including Pipelined-RAM and a processor consistent variant. Maya preprocesses source code to insert Application Programming Interface (API) calls when needed. Since some of the memory consistency models implemented by Maya are too weak to support locks (mutual exclusion), special barrier operations are implemented. The performance of Maya with these weak models was tested with a linear programming application, and some matrix processing algorithms. They found that a performance gain could be attained by weak models after introducing some optimizations to reduce the number of messages sent.

Amza, Cox, Dwarkadas, Keleher, Lu, Rajamony, Yu, and Zwaenepoel created the Treadmarks DSM [36]. Treadmarks provides a semi-transparent DSM interface to the programmer by overriding the page fault handler. It only requires that shared memory be allocated by a special malloc call and that locking and barrier synchronization are performed by Treadmarks API calls. Treadmarks implements the release consistency model. Using a lock call will acquire variables to be modified. Unlocking will trigger the Treadmarks system to create diffs on the Treadmarks memory pages that were accessed. The diff of two memory pages is obtained by subtracting them, location wise. For example:

$$[1,1,1,1,1] - [1,1,5,1,1] = [0,0,-4,0,0]$$

Observe that if most locations do not change, they will contain many runs of 0, making them easily compressible using run length encoding. For example $[1,0,0,0,0,0,0,0,0]$ can be compressed to $[(\textit{run-length}:1,\textit{data}:1),(\textit{run-length}:8,\textit{data}:0)]$. Also observe that diffs that have changes to different locations such as $[1,0,0,0,0]$ and $[0,0,-4,0,0]$ can be combined into a single diff by adding them to obtain $[1,0,-4,0,0]$. Since programs are assumed to protect all shared memory accesses with locks or barriers, Treadmarks can assume that whenever it must process two diffs of the same page, they can be combined in this fashion, simplifying the protocol.

Treadmarks implements two main optimizations to reduce message passing overhead. First it only notifies the next acquirer of the lock that was released of changes. Second an invalidate protocol is used, so diff details are not sent until they are actually needed. Treadmarks was benchmarked using several scientific computing applications.

Maya is most similar to our implementation, however, we use a Message Passing Interface (MPI) system rather than a Parallel Virtual Machine (PVM) one, and provide a plain C++ API interface to the shared memory rather than using a preprocessor. Maya and Treadmarks are more transparent than our application. In our application every assignment must be explicitly performed as a call to a write subroutine. Maya and Treadmarks evaluate performance using scientific computing applications, while we focus our experiments on a lock (mutual exclusion) algorithm.

Mellor-Crummey and Scott evaluated the performance of various lock implementations [42] on various multiprocessor systems. These implementations often relied on strong atomic operations, while the mutual exclusion algorithm we evaluate

uses plain read and write operations. Their concern is the performance of various lock protocols rather than the underlying systems.

2.2 Modeling concurrent systems

To mathematically prove properties of concurrent systems, we must model the systems mathematically. Many frameworks exist to model or specify concurrent systems. The most common types of modeling and specification frameworks are process algebra or automata based.

2.2.1 Process algebra and automata based models

We will illustrate a few process specification frameworks using a running example. This example is similar to the one used by Hoare [34] of a candy machine and a person attempting to purchase toffee from it.

The example system consists of a two processes. One is a candy machine that accepts coins, and dispenses either a toffee or a chocolate, depending on which button is pressed. The other is a person that continually inserts a coin into the machine, chooses toffee by pressing a button, then eats the dispensed toffee.

A popular way of specifying concurrent processes is by using a process algebra. Process algebras build processes by starting with a basic set of processes, then combining them into larger ones using various algebraic operators. Communicating Sequential Processes [34] and the Calculus of Communicating Systems [43] are classic examples.

Communicating Sequential Processes: In Communicating Sequential Processes (CSP) [34] communication is performed by synchronous actions.

The Person part of the system is represented as a recursive process. The identifiers such as *coin* and *toffeebutton* are atomic actions. They have no special meaning (unlike

statements in Hoare logic). On its own, the Person process will repeatedly perform these actions in sequence.

$$\text{Person} \stackrel{\text{def}}{=} (\mathit{coin} \rightarrow \mathit{toffeebutton} \rightarrow \mathit{toffee} \rightarrow \mathit{eat} \rightarrow \text{Person})$$

The candy machine is also represented as a recursive process. The specification of the candy machine contains a choice operator, |, that allows it to choose between the two processes that it is applied to. This choice represents the fact that it is possible to choose the type of candy being dispensed.

$$\begin{aligned}\text{CandyMachine} \stackrel{\text{def}}{=} (\mathit{coin} \rightarrow (\ & \mathit{toffeebutton} \rightarrow \mathit{toffee} \rightarrow \text{CandyMachine} \\ & \mid \mathit{chocolatebutton} \rightarrow \mathit{chocolate} \rightarrow \text{CandyMachine}))\end{aligned}$$

Alone, these processes are not very interesting, but when we compose them, we can see CSP's communication in action.

Combined process:

$$\text{Person} \parallel \text{CandyMachine}$$

The $\parallel$ operator creates a new process that represents the Person and CandyMachine processes communicating and acting in parallel. The CandyMachine and Person must jointly perform the *coin* action. Processes block until shared actions can be performed by all processes at once. For example, the Person process' *eat* action is not shared with any other processes in the composition, so it can be performed alone.

CSP does not distinguish between input and output actions. Intuitively, the person initiates the coin action, but this is not indicated in the CSP specification. This is an elegant, but possibly confusing symmetry. Also, any number of processes can be involved in a single shared action. This makes CSP's shared actions a very strong operation.

Calculus of Communicating Systems: In contrast, the Calculus of Communicating

Systems (CCS) [43] introduced by Milner restricts actions to being either input or output, and only allows two processes to participate in a joint action. Input actions are denoted *action* and output actions are distinguished by an overline $\overline{action}$. Aceto *et. al.* provide a good introduction to CCS theory and its application in Reactive Systems [2]. One of the goals of CCS is to represent concurrency naturally with a minimum of primitives. This simplifies proof mechanics, though it makes some programming tasks a bit more difficult.

The Person process in CCS is very similar to the CSP version, but with the action types distinguished.

$$\text{Person} \stackrel{\text{def}}{=} \overline{coin}.(\overline{toffeebutton}.\overline{eat}.\,\text{Person})$$

The candy machine is also similar, except that the choice operator in CCS is +.

$$\begin{aligned}\text{CandyMachine} \stackrel{\text{def}}{=}& \\ & coin\,.((\,toffeebutton\,.\overline{toffee}.\,\text{CandyMachine}) \\ & \qquad + (chocolatebutton\,.\overline{chocolate}.\,\text{CandyMachine}))\end{aligned}$$

Some CCS presentations use | to denote the parallel composition operator, we use ‖ to maintain consistency with our CSP presentation.

$$\text{Person} \parallel \text{CandyMachine}$$

However, we are not finished, since CCS allows processes to perform output and even input actions without requiring matching actions by another process. This is a significant difference between CCS and CSP. This interesting feature allows us to reason about process equivalence locally. In our example, we only want the CandyMachine to accept choices and dispense goods after a coin has been inserted. This is accomplished by hiding the action with the restriction operator \, which both prevents outside processes from synchronizing with hidden actions, and forces the

processes to perform the hidden actions by synchronization alone.

$$(\text{Person} \parallel \text{CandyMachine}) \setminus \{\mathit{coin}, \mathit{toffee}, \mathit{toffeebutton}, \mathit{chocolatebutton}, \mathit{chocolate}\}$$

The $\overline{eat}$ action is still externally visible, in fact, the only externally visible action of this composed process. We can prove that this composed process is equivalent to the very simple process:

$$\text{Eater} = \overline{eat}.\,\text{Eater}$$

This proof ensures that we can substitute one process for the other in a larger composed system while maintaining its behaviour. This is the type of local reasoning that the design of CCS facilitates.

The π-calculus [44] is an evolution of CCS that supports the representation of mobile processes.

Though CSP, CCS, and π-calculus processes can be analyzed with many different tools, automata based methods are common [2]. These methods represent processes by various types of automata, with states and transitions between states. A system of automata interacting concurrently can be represented as a single non-deterministic automaton. This concrete representation of processes allows a lot of direct mathematical handling, as well as the creation of tools such as model checkers.

Input Output Automata: In the Input Output Automata (IOA) language [41], processes are directly specified as automata. The specified automata communicate by an action synchronization similar to CSP and CCS. Actions in IOA are automata transitions that can arbitrarily modify local state. Their effect on local state is specified in an imperative language that is essentially pseudocode. This can make IOA more

useful for reasoning about algorithms written in imperative programming languages. Actions are designated as input, output, or internal. Each input action is jointly performed with all matching output actions. The input/output matching is similar to CCS, while the fact that more than two processes can participate in a communication action is similar to CSP. The IOA specification of the Person automaton appears in Figure 2.1. The Candy machine automaton is specified in Figure 2.2.

```
automata person
    states   state : enumeration of BUYING,
                                    CHOOSING, EATING
    transitions
      output coin
            pre state = BUYING
            eff state ← CHOOSING
      output toffeebutton
            pre state = CHOOSING
            eff state ← EATING
      input toffee
          eff state ← BUYING
```

Figure 2.1: Person process in IOA. The state must be explicitly tracked and is not automatically maintained by the language as in CSP and CCS.

There is added complexity since we have to explicitly encode the state to represent certain behaviours. In IOA, automata must always accept input events, unlike CSP and CCS processes. This allows the possibility that two coins may be input before a button is pressed. Therefore we have to track the number of coins input. Some other behaviours are allowed (many button presses after coin but before dispensing) and we choose the simplest behaviour.

The composition of the two automata is shown in Figure 2.3. This notation emphasizes that the composition of two automata is another automaton.

```
automata candymachine
    states coinbox : Int
           state : enumeration of COINWAITING,
                   DISPENSETOFFEE, DISPENSECHOCOLATE
    transitions
      input coin
          eff  coinbox ← coinbox +1
      input toffeebutton
          eff if coinbox > 0 then
                state ← DISPENSETOFFEE
      output toffee
           pre state = DISPENSETOFFEE
            eff coinbox ← coinbox − 1
                state ← COINWAITING
      input chocolatebutton
          eff if coinbox > 0 then
                state ← DISPENSECHOCOLATE
      output chocolate
           pre state = DISPENSECHOCOLATE
            eff coinbox ← coinbox − 1
                state ← COINWAITING
```

Figure 2.2: CandyMachine automata in IOA. An IOA automaton must always accept its input actions. Additional logic is needed to handle the case when a button is pushed but no coins have been inserted.

Temporal Logic of Actions: The Temporal Logic of Actions, TLA [39], directly specifies automata using mathematics. It avoids programming languages, making it attractive to hardware designers. TLA also provides many tools for reasoning about automata in general. TLA has been used to reason about automata specified in IOA [45]. TLA provides additional formatting rules and syntax to make formulas easier to read and work with, however, for simplicity we only present the core concepts. We also use elements of our notation and formatting where appropriate for consistency. In this

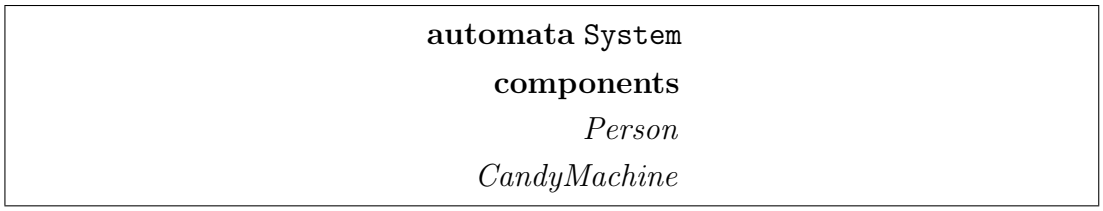


Figure 2.3: Composed system in IOA. Composition is performed using an automaton specification rather than an operator on automatons.

thesis, we emphasize predicates using a special font $\mathsf{predicate}[arg_1, arg_2, \ldots]$, and will also use this convention in our TLA presentation, though it is not standard TLA style.

Transitions (actions) in TLA are specified by two argument predicates defined by expressions involving primed and unprimed variables. The state of a system is a collection of variable assignments.

$$\mathsf{depositcoin} \stackrel{\text{def}}{=} CoinBox' = CoinBox + 1$$

Explicitly this shorthand defines the predicate:

$$\mathsf{depositcoin}[s_1, s_2] \stackrel{\text{def}}{=} s_2.\, CoinBox = s_1.\, CoinBox + 1$$

Predicates defined by expressions that do not involve primed variables are called state predicates.

Unlike the previous frameworks, the names of transitions are simply predicate names, placeholders that make definitions of automata easier to read.

Both types of predicates, actions and state, can be extended to predicates on sequences of states by evaluating them on the first two states in the sequence. They can then be extended to richer predicates on sequences by applying temporal operators. Automata are specified by these predicates on sequences. At the same time, properties and invariants that the automata need to satisfy are also predicates on sequences. This makes TLA very powerful, but also potentially intimidating.

The specification of Person automaton in TLA is given in Figure 2.4 and the Candy Machine automaton is specified in Figure 2.5. $\Box$P asserts that P holds at every suffix of the sequence. Boxing a predicate $[\mathsf{P}]_{<vars>}$ adds in silent (stutter in TLA terminology) transitions, that assert that the variables in *vars* are held constant. Stutter transitions are important when composing automata, since it allows one automaton to stutter while another acts. Applying these operations to the transitions of an automaton gives the predicate representing that automaton. The initial state of the automaton can be specified by using a state predicate which only holds at the beginning of the sequence.

$$
\begin{aligned}
&\mathsf{depositcoin} \stackrel{\text{def}}{=} \mathit{state} = \textsc{buying} \wedge \mathit{CoinBox}' = \mathit{CoinBox} + 1 \\
&\qquad\qquad\qquad \wedge \mathit{wallet}' = \mathit{wallet} - 1 \wedge \mathit{state}' = \textsc{choosing} \\
&\mathsf{makechoice} \stackrel{\text{def}}{=} \mathit{state} = \textsc{choosing} \\
&\qquad\qquad \wedge \mathit{buttonqueue}' = \mathit{buttonqueue}\,.\,\mathrm{append}(\textsc{toffeebutton}) \\
&\qquad\qquad \wedge \mathit{state}' = \textsc{waiting} \\
&\mathsf{eat} \stackrel{\text{def}}{=} \mathit{state} = \textsc{waiting} \wedge \mathit{Dispenser} \neq \emptyset \quad \wedge \mathit{Dispenser}' = \emptyset \wedge \mathit{state}' = \textsc{buying} \\
&\mathsf{InitialPerson} \stackrel{\text{def}}{=} \mathit{state} = \textsc{buying} \\
&\mathsf{PersonTransitions} \stackrel{\text{def}}{=} \mathsf{depositcoin} \vee \mathsf{makechoice} \vee \mathsf{eat} \\
&\mathsf{Person} \stackrel{\text{def}}{=} \mathsf{InitialPerson} \wedge \Box[\mathsf{PersonTransitions}]_{\mathit{wallet}}
\end{aligned}
$$

Figure 2.4: Person automaton in TLA. Communication must be explicitly managed by changing the state of a variable.

Due to its generality, TLA must distinguish between several different types of composition. The main composing operator is conjunction of predicates, however the specifications sometimes have to be tailored to the type of composition desired. Lamport's Specifying Systems [39] provides a discussion of various types of composition possible, including how to obtain a joint action composition similar to the previous frameworks. Composition of the two automata in our example, Figure 2.6, is a bit simpler, since the actions on the shared variables are mutually exclusive. For example,

$$
\begin{aligned}
\mathsf{acceptcoin} &\stackrel{\text{def}}{=} \neg\, coinaccepted \wedge CoinBox \geq 0 \\
&\quad \wedge\, CoinBox' = CoinBox - 1 \\
&\quad \wedge\, coinaccepted' \wedge buttonqueue' = () \\
\mathsf{dispenseToffee} &\stackrel{\text{def}}{=} buttonqueue\,.\,\text{not-empty}() \wedge coinaccepted \\
&\quad \wedge\, buttonqueue\,.\,\text{head}() = \textsc{toffeebutton} \\
&\quad \wedge\, Dispenser' = Dispenser\,.\,\text{append}(\textsc{toffeebar}) \\
\mathsf{dispenseChocolate} &\stackrel{\text{def}}{=} buttonqueue\,.\,\text{notempty}() \wedge coinaccepted \\
&\quad \wedge\, buttonqueue\,.\,\text{head}() = \textsc{chocolatebutton} \\
&\quad \wedge\, Dispenser' = Dispenser\,.\,\text{append}(\textsc{chocolatebar}) \\
\mathsf{InitialCandyMachine} &\stackrel{\text{def}}{=} \neg\, coinaccepted \wedge CoinBox = 0 \wedge buttonqueue = () \\
\mathsf{CandyMachineTransitions} &\stackrel{\text{def}}{=} \mathsf{acceptcoin} \vee \mathsf{dispenseToffee} \vee \mathsf{dispenseChocolate} \\
\mathsf{CandyMachine} &\stackrel{\text{def}}{=} \mathsf{InitialCandyMachine} \wedge \Box[\mathsf{CandyMachineTransitions}]_{coinaccepted}
\end{aligned}
$$

Figure 2.5: Candy Machine in TLA

$$
\mathsf{InitCandyMachine} \wedge \mathsf{InitPerson} \wedge \Box[\mathsf{CandyMachineTransitions} \vee \mathsf{PersonTransitions}]_{<vars>}
$$

Figure 2.6: Composition of Person and Candy Machine in TLA

if we added another Person to the composition then simply composing would yield a transition where only one coin was deposited but both persons moved to the choosing state. Even in this simpler case, one system cannot stutter on a shared variable while the other acts on it, so we must perform a more involved composition using the components of the specifications.

2.2.2 Memory consistency modeling

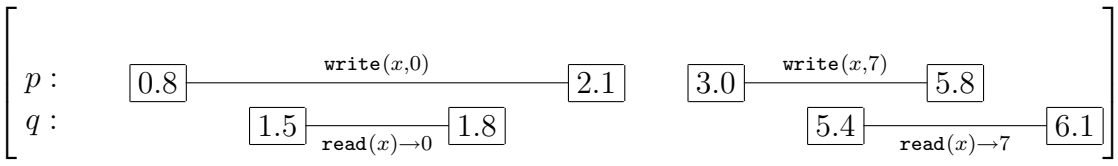
Memory consistency modeling, mentioned in the introduction, allows us to model the aspects of memory systems that affect program correctness without also modeling the implementation details of these memory systems.

There is a large body of literature examining various memory consistency models. Steinke and Nutt [50] provide references to many memory consistency models and provide unified way to define them. This thesis will focus on the sequential consistency, Pipelined-RAM and processor consistency memory models.

Lamport introduced one of the first models of shared memory to reason about their low level implementations. In this model, the System Executions framework [38] with global time, we are concerned about the operations that processes using the shared memory can perform. The shared memory is a collection of objects that are operated on individually.

We reason about systems in this model by considering the observable behaviour of the system. In this thesis, representations of observed behaviour are called *computations*. In the literature, such representations are sometimes called histories, behaviours, or traces.

In Lamport's framework, computations associate operations with the intervals of time in which they execute. An example of such a computation in the framework is:



In this example, the concurrent read and write operations by different processors overlap in time. This could be a problem for a hardware implementation. In binary $7 = 111_2$, so changing the value of x requires changing 3 bits individually. Depending on the implementation, it could be possible for the read of x to return any 3 bit value.

This is a problem for any process using such a shared memory. Lamport approaches this problem by defining classes of shared memories that differ on the values that a concurrent read can return. We present the regular and atomic shared memory

definitions introduced by Lamport [38]. Both assume that there are no concurrent writes to any variable, so the writes to a variable are totally ordered and separate in time.

Definition 2.2.1. *A* regular shared memory *is one in which each read of a variable x returns either:*

1. *the value of a concurrent write to x*
2. *the value of the most recent preceding write to x*

Definition 2.2.2. *An* atomic shared memory *is a shared memory that is regular and in addition, one in which no read of a variable x returns the value of a write that happened before the write returned by a previous read.*

The definition of atomic shared memory is tied to shared read/write variables and is quite subtle. In general we would like to consider many types of concurrent objects: concurrent queues, concurrent lists, concurrent dictionaries, and more. Herlihy and Wing generalized the definition of atomic registers in an expanded framework to cover general data types and eliminate the concurrent write restriction. We will call this framework the global time framework. In this framework, Herlihy and Wing define their generalization of atomic registers, called linearizability.

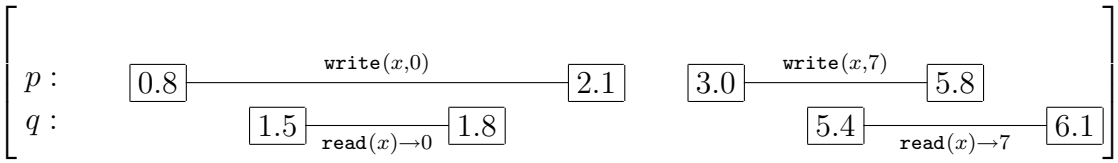
The global time framework represents operations as an invocation and a response. The timestamp of the invocation and response of an operation form an interval of time that covers the execution of the operation. To allow any object to be covered by the definition, we allow any objects that can by specified by a sequential specification. The sequential specification of an object is the collection of sequences of operations for an object that are considered valid. Sequential specification is described in more detailed in Chapter 3. The combination of these modeling techniques allows the definition of Linearizability to be more general and less complex at the same time. Linearizability can be defined in the following manner:

Definition 2.2.3. *A shared memory is linearizable if for each execution, we can assign each operation a unique time, called its* linearization point *in its interval such that:*

The sequence of operations formed by ordering the operations by their linearization points is valid (satisfies the sequential specification of the objects).

Our presentation defines atomicity and linearizability as properties of the memory as a whole. In most of the literature, these properties are defined as properties of the *objects*. Rather than saying that a memory is atomic, these definitions define an object as atomic. A result by Herlihy and Wing justifies this terminology: a collection of linearizable shared memories, each with a single object (a collection of linearizable objects in standard terminology), together form a linearizable shared memory. This result does not hold in weaker memory models, so we must reason about memories that manage objects, rather than objects in isolation.

Sequential consistency [37] lacks the global time required by Linearizability, but still agrees with the programmer's intuition of how a shared memory should behave. To specify sequential consistency, we move to local time computations. Local time computations contain no information about the global time properties of operations. Further the local time is only represented by a per process total order, rather than time values. In a local time representation, our previous example:



Would lose much of its information in this model, and its representation would be:

$$\begin{bmatrix} p: & \texttt{write}(x,0)\ \texttt{write}(x)7 \\ q: & \underline{\texttt{read}(x)}\ \underline{\texttt{read}(x)} \\ & 0 \quad\ 7 \end{bmatrix}$$

We have significantly reduced the information represented in the model. This has the advantage that the underlying implementation of the shared memory is given more freedom in how operations are performed. Given this representation, we can now define sequential consistency. A computation C is sequentially consistent if there is a valid total order of all the operations of the computation that agrees with the local time order (known as program order). Recall that a valid total order is one that matches the sequential specification of the objects in the system.

To the programmer, sequential consistency and linearizability are very similar. Sequential consistency has a definite advantage over linearizability, that was proven by Attiya and Welch [8]. Their result presents a sequentially consistent replicated shared memory with a guaranteed message delay that is less than the lower bound message delay for a linearizable replicated shared memory.

Lipton and Sandberg [40] introduced the Pipelined-RAM memory model. This memory model can be defined using only local time. It is natural from the shared memory implementer's point of view, but not the programmer's. Unlike sequential consistency it does not require a shared global total order view on all operations to be shared by all processors. We call memory models that are weaker than sequential consistency, *weak memory models*. Mutual exclusion cannot be solved on Pipelined-RAM systems with only read/write variables [31].

Various forms of the processor consistency memory consistency model were created based on Goodman's informal definition [25]. Ahamad *et. al.* provided one formalization [4] and proved some properties about its support for standard mutual exclusion algorithms. We will refer to the formalized model of Ahamad *et. al.* as PC-G. Since several mutual exclusion algorithms are supported by PC-G, it is one of the few weak models that can be used to implement sequential consistency with only read/write variables. Higham, Kawash and Verwaal [33] explore the various definitions of processor consistency, concluding that many are in fact incomparable. Higham and

Kawash have explored implementations of sequential consistency on PC-G with mutual exclusion [31] and satisfying wait-free progress guarantees [32].

The previously mentioned models are idealized and simplified. As multithreaded software becomes more prevalent, it is increasingly important to study the memory consistency models of widely available microprocessors. Higham, Jackson, and Kawash explore memory consistency models for SPARC [30] and Itanium [28, 29] microprocessors. The consistency model for Alpha processors, described in the Alpha manual [20], was formally defined and investigated by Attiya and Friedman [6]. The PowerPC consistency model was formalized by Corella, Stone, and Barton [17] . x86 architecture has not been formalized to our knowledge. The Intel architecture developer manual provides some description of an x86 memory model [19] which is also clarified in an Intel whitepaper [18].

2.3 Proofs of correctness of concurrent systems

Frameworks for proving the correctness of computer systems first dealt with the correctness of sequential programs. For programs written in imperative sequential languages, Hoare logic forms a basis for proofs of correctness.

2.3.1 Hoare logic based proof methods

Hoare logic provides a solid foundation for proving the correctness of sequential programs. Backhouse [10] provides a good textbook introduction to Hoare logic. The basic element of Hoare logic is the Hoare triple. A Hoare triple $\{\mathsf{P}\}S\{\mathsf{Q}\}$ consists of a statement S, preassertion P and postassertion Q. A Hoare triple is *correct* depending on the semantics of the statement S. For example, the Hoare triple $\{x = 5\}x \leftarrow x + 1\{x = 6\}$ is correct, while $\{x < 4\}x \leftarrow x + 1\{x = 6\}$ is not.

Correct Hoare triples $\{\mathsf{P}\}S\{\mathsf{Q}\}$ and $\{\mathsf{Q}\}T\{\mathsf{R}\}$ with a matching assertion $\{\mathsf{Q}\}$ can be

connected, forming a correctly annotated program $\{\mathsf{P}\}S\{\mathsf{Q}\}T\{\mathsf{R}\}$. To prove the correctness of a sequential program using Hoare logic, a program is first specified by providing a preassertion and a postassertion. An assertion is a predicate that is used to ensure that certain properties hold at a point in the execution of a program. A program is correct if we can correctly annotate the program, connecting the preassertion to the postassertion.

What is less widely appreciated is that Hoare logic also provides a method of program construction. This allows a program to be derived from its specification, building the proof of correctness as it is constructed. Backhouse's textbook [10] also provides an introduction to this program construction method.

Hoare logic was generalized to concurrent shared variable multiprograms by Owicki and Gries. We adapt the presentation from Feijen and van Gasteren's book [22]. In this presentation, a multiprogram consists of several sequential *components* that are executed concurrently and asynchronously. This presentation splits the Owicki-Gries notion of correctness into two forms, local and global correctness. The components of multiprograms are annotated with Hoare triples, and the annotations must satisfy the local and global correctness conditions to form a correctly annotated multiprogram.

Definition 2.3.1. *The* local correctness *of an assertion is defined separately for initial and non-initial assertions.*

1. *An initial assertion* $\{\mathsf{P}\}$ *of a component is* locally correct *if it is implied by the preassertion of the multiprogram as a whole.*
2. *A non-initial assertion* $\{\mathsf{P}\}$ *must be textually preceded by some* $\{\mathsf{Q}\}S$*. It is* locally correct *if* $\{\mathsf{Q}\}S\{\mathsf{P}\}$ *is a correct Hoare triple.*

This is the standard correctness notion for sequential programs, each component examined individually must consist of correct Hoare triples. We must then add a rule to consider the interleavings of the atomic statements of all the components. For each

assertion in a component, we must prove that the assertion is maintained by all of the atomic statements of the other components.

Definition 2.3.2. *An assertion* $\{\mathsf{P}_p\}$ *in a component* p *is* globally correct *if for each* $\{\mathsf{Q}_q\}S_q$ *in every other component* q*:*

$$\{\mathsf{P}_p \wedge \mathsf{Q}_q\}S_q\{\mathsf{P}_p\}$$

is a correct Hoare triple.

For many specifications, annotations must be augmented to provide the proofs of correctness. Often we cannot simply use the natural ones provided by the definition of the statements. The trick is then finding the proper annotations to satisfy the global and local correctness rules.

Rely/guarantee reasoning, introduced by Jones (a recent presentation is [15]) offers a strategy to simplify the proof of global correctness. Observe that in a two process system, with components p and q, each having p_N and q_N atomic statements, we must perform roughly $p_N \times q_N$ proofs for each component.

Consider a system consisting of the following annotated components:

Component p

$\{x \geq 0\}$
1 $x \leftarrow x + 1$
$\{x \geq -1\}$
2 $x \leftarrow x + 2$
$\{x \geq -2\}$
3 $x \leftarrow x + 3$
$\{x \geq -3\}$

These annotations are deliberately weakened from the natural ones. In general we must perform such weakenings to allow for the interaction of the other processes. For example, a more natural annotation of $x \leftarrow x + 1$ would be $\{x \geq 0\}x \leftarrow x + 1\{x \geq 1\}$.

However, the weakened annotation can cope with the possibility that q decreases the value of x.

Component q
$\{x \geq 0\}$
1 $x \leftarrow x \times 5$
$\{x \geq 0\}$
2 $x \leftarrow x \times 6$
$\{x \geq 0\}$
3 $x \leftarrow x \times 7$
$\{x \geq 0\}$

Consider the global correctness proof for p, in this example, it requires 16 proofs. Suppose that there is a predicate R_p such that for all statements S_p in p, if $\{\mathsf{P}_p\}S_p$ is an annotation, then $\mathsf{R}_p \implies \mathsf{P}_p$. In this example, a natural choice is $\mathsf{R}_p = x \geq 0$. We then only have to prove the global correctness of R_p, needing only q_N proofs. In this example, we reduce the 16 proofs to 4 that are easily verified. To reduce this number further, we may find a predicate G_q, such that for all atomic statements $\{\mathsf{P}_q\}S_q$ in q, $\{\mathsf{G}_q \wedge \mathsf{P}_q\}S_q\{\mathsf{G}_q\}$ is a correct Hoare triple. The natural choice in this example is $\mathsf{G}_q = x \geq 0$. It may then be possible to prove the global correctness of p simply by proving $\mathsf{G}_q \implies \mathsf{R}_p$, requiring only one proof. We call R_p the *rely* assertion of p and G_q the *guarantee* assertion of q. Proof of a multiprogram using rely/guarantee techniques will require the invention of a rely and a guarantee assertion for each process.

Separation logic is another extension of Hoare logic that allows reasoning about the structures created in memory by pointer (memory address) references. This is done by introducing assertions about the separation of various parts of the heap in terms of pointer reachability. For example, consider the following array copying routine, similar to a memcpy in C. Here we use the notation $[x]$ from separation logic to denote pointer dereferencing.

arraycopy($array_1, array_2, length$)
1 **for** $i \leftarrow 0$ **to** $length - 1$
2 **do** $[array_2 + i] \leftarrow [array_1 + i]$

In the C programming language, it is possible that the two ranges of addresses $[array_1, array_1 + length - 1]$ and $[array_2, array_2 + length - 1]$ overlap. This could cause the copying routine to produce an undesired result.

Dealing with such pointer aliasing situations is quite difficult but necessary to prove correctness. Separation logic [47] introduces new logical operators, the *separating conjunction* $*$, and *separating implication* $-\!\!*$ to allow easier reasoning about heap structure, with new statements that can be used to manipulate the heap. We show how the separating conjunction can be used with the array copying example, to give an idea of how reasoning proceeds. We would like to formally ensure that this procedure is only used when the arrays do not share memory.

In separation logic, we can ensure this with the preassertion :

$$\begin{aligned}\{&(\forall i \in 0, \ldots, length - 1 : (array_1 + i) \mapsto -) \\ &* (\forall i \in 0, \ldots, length - 1 : (array_2 + i) \mapsto -)\}\end{aligned}$$

Both conjuncts, $(\forall i \in 0, \ldots, length - 1 : (array_1 + i) \mapsto -)$ and $(\forall i \in 0, \ldots, length - 1 : (array_2 + i) \mapsto -)$ simply state that all the locations of each array have some undetermined value, a very weak assertion. However the separating conjunction of the two assertions asserts that the addresses referred to in each of the statements are disjoint, which is exactly what we want. We may then note that this assertion is preserved by the procedure, so we may correctly call this procedure with the same arguments several times in a row, though it would not produce any changes. This would not be true of more complicated assertions and procedures with a lot of pointer manipulation, such as removing an element from a doubly linked list.

Concurrent separation logic [12] is an extension of separation logic that allows reasoning about concurrency in an Owicki-Gries style. As most data structures involve pointer linked structures, it would be very useful to apply these techniques in a memory consistency setting.

Program proofs require many tedious and subtle steps of reasoning. Formalizing

the proofs can reduce and expose errors early on. Calculational proof, introduced by Dijkstra, Feijen, and Scholten, is a useful formal tool for reasoning with Hoare logic. Rather than providing proofs in a mixture of natural language and mathematical notation, calculational proof presents proofs as a series of formal manipulations of logical statements. The formality of the proof may be relaxed as needed, but the format naturally encourages small, easily justified steps in reasoning. An informal calculational proof of a property of integer divisors, based on a proof of Euclid's gcd algorithm in [26], is:

$$
\begin{aligned}
& (p > q) \wedge \mathsf{divides}[d, p - q] \wedge \mathsf{divides}[d, q] \\
\Longrightarrow\ & \{\text{definition of } \mathsf{divides}\} \\
& (p - q) = m \times d \wedge q = n \times d \\
\Longrightarrow\ & \{\text{arithmetic}\} \\
& p = (p - q) + q = m \times d + n \times d = (m + n) \times d \\
\Longrightarrow\ & \{\text{definition of } \mathsf{divides}\} \\
& \mathsf{divides}[d, p]
\end{aligned}
$$

Structured calculational proof [9], is an extension of Dijkstra, Feijen and Scholten's calculational proof that allows nesting of proofs. This is helpful for isolating pieces of proofs without creating additional lemmas. Calculational proofs are very useful for providing proofs in proof frameworks based on Hoare logic. Lamport provides a few calculational proofs of properties of the Temporal Logic of Actions (TLA), a different proof framework, in Specifying Systems [39]. This thesis occasionally uses calculational style to present proofs.

2.3.2 Automata based proof methods

Proofs usually involve some combination of induction over automata behaviour and simulations between automata. CCS and π-calculus processes are proved equivalent by proving a similarity between their automata [2]. In TLA inductive arguments [1] are usually used to show that automata implement specifications by showing that

automata $\Longrightarrow$ spec. Simulation is useful when the framework allows automata to be specified at a much higher level than can be directly implemented. Automata based methods are also useful in that they offer the possibility of automated verification. Programs called model checkers [2] can explore the state space of an automaton, exhaustively testing desired properties.

2.3.3 Non-automata based proof methods

Lamport's System Executions [38] framework is used to prove the correctness of concurrent systems in a model that only assumes partially ordered time. A system execution is a triple $(S, \rightarrow_B, \rightsquigarrow_B)$ with a set of events S, happens-before order $\rightarrow_B$ and can-causally-affect order $\rightsquigarrow_B$ that satisfy the system execution axioms. One can view a system execution as modeling an underlying partially ordered set $(A, \rightarrow_B)$. Given this partially ordered set, we can define two relations induced on the subsets of A. Let C, D be subsets of A then define $C \rightarrow_B D \stackrel{\text{def}}{=} \forall c \in C, d \in D : c \rightarrow_B d$ and $C \rightsquigarrow_B D \stackrel{\text{def}}{=} \exists c \in C, d \in D : c \rightarrow_B d$. The first is a partial order, while the second is not. We take a similar approach of using a partial order on a set to induce various relations on its subsets for our proofs. The System Execution axioms allow us to reason about system executions independent of this intuitive model, allowing us to reason about the system without knowing the exact partial order of time, which may be attractive in certain applications. Weak memory consistency models usually involve more than one partial order, so System Executions cannot be directly used to reason about them. However, the System Executions framework provides many useful techniques for manipulating partial orders that are easily generalized.

Gischer [24] and Pratt [46] define a process algebra of labeled partially ordered sets (pomsets) along with an equational theory. We define a notion of transformation of traces (Section 3.2.3) based in part on the definitions of the substitution and parallel composition operators of this process algebra. As with the System Executions

framework, the pomset framework only assumes partially ordered time.

Many proofs of the lazy sequential consistency protocol in various frameworks are provided in an issue of Distributed Computing [23]. This protocol is more similar to a hardware cache protocol rather than a message passing one. The goal of these proofs are most similar to ours, implementing a shared memory on a lower level system.

Many proofs of shared memory algorithms have assumed a memory consistency model introduced by Herlihy and Wing, Linearizability [27]. Unlike many memory consistency models, Linearizability assumes a strong notion of global time discussed in Section 2.2.2. Since this strong assumption admits many different proof techniques, it is a popular starting point for proving the correctness of shared memory algorithms. However, proofs that make heavy use of strong global time are difficult to migrate to weaker models. Recently, Vafeiadis, Herlihy, Hoare and Shapiro have developed a new method of proving the correctness of linearizable data structures [51]. Their technique merges the rely/guarantee method described earlier, the linearization point proof method described in Section 2.2.2, and linking invariants, a technique used to prove the correctness of implementations of abstract data types. This combined method is used to prove the correctness of a highly concurrent linked list algorithm.

Herlihy and Wing introduce a useful technique for modeling shared memory in their Linearizability paper [27]. This thesis uses an adaptation of their technique for its modeling. Proofs of distributed shared memory implementations usually rely on the automata based methods described earlier. In these proofs, the shared memory implementation is modeled operationally as an automaton, then it is shown that the behaviours of this automaton match the specified memory consistency model. This thesis models both the specification system and target implementation system using memory consistency models, allowing a uniform presentation of proofs. Aspinal *et. al.* use a similar style of modeling to formalize the Java Memory Model (JMM) [5] and provide machine verified proofs. This work of Aspinal *et. al.* focuses on the foundations

of the JMM, such as the behaviour of a certain class of programs on a JMM system. This thesis presents general techniques for reasoning about programs running on systems with weak memory models.

2.4 Relation between thesis and literature

We present proofs of correctness of concurrent systems based on memory consistency models. This differs from most of the automata based methods currently in use. The proof techniques presented in this thesis apply to weak models with local time, rather than requiring strong global time assumptions.

We provide several replicated implementations of shared memory. These are implemented using two different strategies. One uses a token algorithm to enforce mutual exclusion. The other uses timestamps and priority queues to synchronize the order of operations. These strategies are well known, but we use them to implement memory consistency models that have not been previously implemented.

This thesis introduces a generalization of the PC-G memory model, Partition Consistency, that is parameterized. Choosing the parameter allows us to obtain other memory consistency models, such as sequential consistency and Pipelined-RAM as special cases. This means that our correct implementations of partition consistency are also correct implementations of its special cases.

To prove the correctness of our implementations, we created memory consistency models with threads, broadcast primitives, and network primitives. We believe that these models demonstrate the broad applicability of memory consistency reasoning techniques.

Chapter 3

Modeling Distributed Shared Memory

In this thesis, a distributed shared memory system consists of a set of processes, operating on a set of objects in memory. The details of how the distributed shared memory system manages the operations performed by the processes on the objects is a predicate called a memory consistency model. In this thesis, predicates will be emphasized by a special font $\mathsf{predicate}[arg_1, arg_2, \ldots]$. As discussed in Sections 1.1 and 2.2.2, a memory consistency model abstracts the implementation details of the memory system, leaving the properties necessary to prove correctness. Formally, we define a *(multiprocess) system* to be a triple of the form (P, J, MC) where P denotes a collection of *processes* operating on a set of *objects*, J, under a memory consistency model MC.

Review of relations and partial orders: A *relation* R between sets A and B is a set of pairs (a, b) where $a \in A$ and $b \in B$. Relations will be an essential element of our modeling and proof techniques. We will emphasize relations using the notation $\underset{R}{\longrightarrow}$, $\underset{R}{\models}$, and $\underset{construct}{\sim}$ depending on the relation's application. Relations discussed here are mostly between a set A and itself. We can define relations by providing the set of pairs by directly specifying it

$$\underset{R}{\longrightarrow} \stackrel{\text{def}}{=} \{(x, y) : \mathsf{predicate}[x, y]\}$$

or building them out of the standard set operators. We also use a shorthand

$$x \underset{R}{\longrightarrow} y \stackrel{\text{def}}{=} \mathsf{predicate}[x, y]$$

that can be more convenient.

The inverse of a relation $\underset{R}{\longrightarrow}$ is denoted $\underset{R^{-1}}{\longrightarrow}$:

$$\underset{R^{-1}}{\longrightarrow} \stackrel{\text{def}}{=} \left\{(b,a) : (a,b) \in \underset{R}{\longrightarrow}\right\}$$

For a relation $\underset{R}{\longrightarrow}$ its *strict transitive closure*, $(\underset{R}{\longrightarrow})^+$ for our purposes can be defined inductively:

$$\begin{aligned}
(\underset{R}{\longrightarrow})^1 &= \underset{R}{\longrightarrow} \\
(\underset{R}{\longrightarrow})^n &= \left\{(x,z) : \exists y : x(\underset{R}{\longrightarrow})^{n-1} y \wedge y \underset{R}{\longrightarrow} z\right\} \text{ if } 1 < n \in \mathbb{N} \\
(\underset{R}{\longrightarrow})^+ &= \bigcup_{n=1,2,3,\ldots} (\underset{R}{\longrightarrow})^n
\end{aligned}$$

The notion of an *extension* of a relation, defined below, will be essential to define the necessary memory consistency models. It is similar to one relation being a subset of another, but we only consider the pairs in the relation that have elements in an arbitrary set A.

$$\mathsf{Extends}[\underset{R}{\longrightarrow}, \underset{T}{\longrightarrow}, A] \stackrel{\text{def}}{=} \forall a_1, a_2 \in A : a_1 \underset{T}{\longrightarrow} a_2 \implies a_1 \underset{R}{\longrightarrow} a_2$$

Agreement of relations is similar to equality of relations, but like extension, incorporates a restriction:

$$\mathsf{Agree}[\rightarrow_R, \rightarrow_T, A] \stackrel{\text{def}}{=} \forall a_1, a_2 \in A : (a_1 \underset{R}{\longrightarrow} a_2) \Leftrightarrow (a_1 \underset{T}{\longrightarrow} a_2)$$

A *strict partially ordered set* is a pair $(A, \underset{R}{\longrightarrow})$ that satisfies transitivity and irreflexivity on A:

$$\left(\forall a_1, a_2, a_3 \in A : (a_1 \underset{R}{\longrightarrow} a_2 \wedge a_2 \underset{R}{\longrightarrow} a_3) \implies a_1 \underset{R}{\longrightarrow} a_3\right) \wedge \left(\forall a \in A : \neg(a \underset{R}{\longrightarrow} a)\right)$$

In this thesis, we will refer to strict partially ordered sets as *partial orders*.

A *strict totally ordered set* is a pair $(A, \underset{R}{\longrightarrow})$ that is a partial order and satisfies the property that $\forall a, b \in A : (a \underset{R}{\longrightarrow} b) \vee (b \underset{R}{\longrightarrow} a) \vee a = b$. In this thesis, we will refer to strict totally ordered sets as *total orders*. A finite total order naturally induces a sequence, and a finite sequence, with no repeated elements, naturally induces a total order.

Model of objects: Each object in J is defined by stating the possible operation invocations that can be applied to it, and then providing its *sequential specification* [27]. Call the combination of an operation invocation and response a *completed operation*. The sequential specification of an object specifies the sequences of completed operations that are considered *valid*. The valid sequences of operations are those that could occur in a sequential execution. In this thesis, we will specify these sequences by directly describing them.

Completed operations that require a response will be denoted as $\dfrac{\texttt{opname}(arg_1, arg_2, \dots)}{response}$ and those that do not will be denoted as $\texttt{opname}(arg_1, arg_2, \dots)$.

A *read/write variable* is an object x that supports the operations $\texttt{read}(x)$ and $\texttt{write}(x, v)$ and its sequential specification is the set of all sequences of $\dfrac{\texttt{read}(x)}{v}$, and $\texttt{write}(x, v)$ such that each $\texttt{read}(x)$ in the sequence returns the value of the most recent preceding $\texttt{write}(x, v)$ or the initial value of the object if there is no preceding write.

Examples of valid sequences for this object are:

$$\left[\ \texttt{write}(x, 5), \frac{\texttt{read}(x)}{5}\ \right]$$

$$\left[\ \texttt{write}(x, 5), \texttt{write}(x, 6), \frac{\texttt{read}(x)}{6}\ \right]$$

$$\left[\ \frac{\texttt{read}(x)}{0}\ \right] \text{ assuming 0 is the initial value of x}$$

Model of processes: Each process in a multiprocess P consists of one or more threads. We will assume that the processes in P are numbered starting at 0. Each

thread performs local computations and issues a sequence of operation invocations on the objects in J. Threads are specified by code. The pair (P, J) is called a *(multiprocess) program*. This distinction allows us to consider the effect of different memory consistency models on the same program. An example of a multi threaded multiprocess is:

$$P_{MTex} \stackrel{\text{def}}{=} \begin{cases} p.m: & \mathit{theAnswer} \leftarrow 42; \mathit{answerReady} \leftarrow \text{TRUE}; \\ p.s: & \textbf{while } \neg \mathit{answerReady} \ \{\textbf{skip};\} \ \texttt{send}(p, q, [\text{MSG}, \mathit{theAnswer}]) \\ q.m: & ([\text{MSG}, \mathit{recvAnswer}]) \dashv \text{recv}(); \text{print}(\mathit{recvAnswer}) \end{cases}$$

This process is meant to be executed on a platform that we will introduce later, the threads of a process p communicate with each other by shared memory. The processes p and q can only communicate with each other by sending and receiving messages. This models networked systems of multiprocessor nodes, that have similar communication configurations.

For single threaded multiprocesses (multiprocesses in which every process has exactly one thread), we omit the thread names:

$$P_{STex} \stackrel{\text{def}}{=} \begin{cases} p: & \mathit{theAnswer} \leftarrow 42; \mathit{answerReady} \leftarrow \text{TRUE}; \\ q: & \textbf{while } \neg \mathit{answerReady} \ \{\textbf{skip};\} \\ & \mathit{qAnswer} \leftarrow \mathit{theAnswer}; \mathit{qReady} \leftarrow \text{TRUE} \\ r: & \textbf{while } \neg \mathit{qReady} \ \{\textbf{skip};\} \ \text{print}(\mathit{qAnswer}); \end{cases}$$

Model of traces: In most models of concurrency, a trace is a record of the actions taken by the concurrent processes and is usually a single sequence of these actions. This assumes a strong global notion of time. Our models also represent records of the actions taken by the concurrent processes, but they are partially ordered, so we call these records *computations* rather than traces.

An *individual (thread) computation* for a thread $p.t$ of a process, p, is formed by *arbitrarily* completing each operation invocation on a shared object in p's invocation sequence with a response value. For an individual computation I, O_I will denote the set of completed operations in I. An individual computation I is *compatible* with a set of objects J if all the operations in O_I are operations on objects in J. A sequence defines a total order on its elements, for an individual computation I, $(O_I, \underset{\text{prog}_I}{\longrightarrow})$ denotes this total order, called the *program order* of the individual computation.

The notation $\langle\ x_a : a \in A : \mathsf{pred}[x_a]\ \rangle$ specifies a collection of items x_a, one for each $a \in A$, such that pred holds for each of them.

A *(multiprocess) computation* of the multiprocess program (P,J) is a collection of individual computations compatible with J, one for each thread $p.t$ in each process $p \in P$. Using our notation, a computation of the multiprocess program (P,J) is a collection $\langle\ I_{p.t} : p.t \in p \in P : I_{p.t}$ individual computation compatible with $J\ \rangle$

The set of operations O_C of a computation is the union of the operations of its individual computations. The *program order*, $\underset{\text{prog}_C}{\longrightarrow}$, of a computation is the union of the disjoint program orders of its individual computations. The program order is a partial order on all operations O_C of a computation C. This partial order is denoted $(O_C, \underset{\text{prog}_C}{\longrightarrow})$. When the computation being referenced is clear, we will omit the subscripts and simply write $(O, \underset{prog}{\longrightarrow})$.

An example of a computation of the example multiprogram (P_{MTex}, J_{MTex}) is

$$C = \left[\begin{array}{ll} \widehat{p}.m: & \texttt{write}(\mathit{theAnswer}, 42), \texttt{write}(\mathit{answerReady}, \text{TRUE}) \\ \widehat{p}.s: & \dfrac{\texttt{read}(\mathit{answerReady})}{\text{TRUE}}, \dfrac{\texttt{read}(\mathit{theAnswer})}{42}, \texttt{send}(\widehat{p}, \widehat{q}, [\text{MSG}, 42]) \\ \widehat{q}.m: & \dfrac{\texttt{recv}()}{\widehat{p}, \widehat{q}, [\text{MSG}, 42]}, \text{print}(42) \end{array}\right]$$

with

$$(O_C, \underset{\text{prog}_C}{\longrightarrow}) = \left(\begin{array}{l} \texttt{write}(\textit{theAnswer}, 42) \underset{\text{prog}_C}{\longrightarrow} \texttt{write}(\textit{answerReady}, \text{TRUE}) \\ \dfrac{\texttt{read}(\textit{answerReady})}{\text{TRUE}} \underset{\text{prog}_C}{\longrightarrow} \dfrac{\texttt{read}(\textit{theAnswer})}{42} \underset{\text{prog}_C}{\longrightarrow} \texttt{send}(\widehat{p}, \widehat{q}, [\text{MSG}, 42) \\ \dfrac{\texttt{recv}()}{\widehat{p}, \widehat{q}, [\text{MSG}, 42]} \underset{\text{prog}_C}{\longrightarrow} \text{print}(42) \end{array} \right)$$

The notation $\mathcal{C}(P, J)$ will denote the set of all computations generated by a program (P,J). For computations of single threaded multiprograms, we will omit the thread names as we did for their process specifications.

Model of memory consistency: Notice that the preceding definitions do not restrict the values that are returned by operations on shared objects in a computation of the program (P, J). The possible responses are determined by the architecture of the system. This architecture is captured by the memory consistency model, MC. It is specified by defining a predicate on computations.

In this paper, the predicate will take the form of two types of requirements:

[**VTO** requirements:] asserts the existence of a collection of valid total orders on the operations of the computation that must extend certain partial orders.

The memory consistency models that will be defined will require exactly one valid total order per process.

[**AGR** requirements:] a collection of agreement properties that the **VTO** total orders must satisfy.

Orders that satisfy **[VTO]** and **[AGR]** are called *witness orders*. If C is a computation, and X is a collection of orders that satisfy the requirements of $\mathsf{MC}[C]$, then we say that X is a witness to $\mathsf{MC}[C]$.

The *computations of the system* (P, J, MC) are all the computations of the program (P, J) that satisfy the constraints of MC. The pair (J, MC) is called a *platform*.

The *computations of a system* are the computations of its program that satisfy its memory consistency model. We denote the computations of a system by the notation $\mathcal{C}(P, J, \mathsf{MC})$.

$$\mathcal{C}(P, J, \mathsf{MC}) \stackrel{\text{def}}{=} \{C : C \in \mathcal{C}(P, J) \wedge \mathsf{MC}[C]\}$$

In order to refer to subsets of a set of operations we introduce the following notation. $O|x$ where x is a variable name, will refer to the subset of operations in O on x. $O|\mathsf{writes}$ is the subset of operations in O that are writes to variables. $O|\mathsf{writes}(S)$ is the subset of writes to a variable x where $x \in S$. $O|p$ where p is a process, is the subset of operations invoked by p. The notation $proc(o)$ denotes the process that invoked an operation o.

3.1 Models of specification and implementation platforms

We first introduce some standard memory consistency models from the literature. Sequential consistency was first defined by Lamport [37] which can be formalized as follows.

$$\begin{aligned}\mathsf{SequentialConsistency}[C] \stackrel{\text{def}}{=} [\mathbf{VTO}]\exists \langle\, (O, \underset{L_p}{\longrightarrow}) \text{ valid total order} : p \in P \\ : \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O\,\rangle : \\ [\mathbf{AGR}]\forall p, q \in P : \mathsf{Agree}[\underset{L_p}{\longrightarrow}, \underset{L_q}{\longrightarrow}, O]\end{aligned}$$

This definition is equivalent to requiring a single total order, since the agreement conditions force all of the views to be equal. This simplification is used in the network

framework model, we don't reason about local views of each individual thread.

An equilvalent definition from Higham and Kawash [32] is:

$$\begin{aligned}\mathsf{SequentialConsistency}[C] \stackrel{\text{def}}{=} [\mathbf{VTO}]\exists \langle\ (O|p \cup O|\mathsf{writes}, \underset{L_p}{\longrightarrow}) \text{ valid total order} : p \in P \\ : \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O|p \cup O|\mathsf{writes}]\ \rangle : \\ [\mathbf{AGR}]\forall p, q \in P : \mathsf{Agree}[\underset{L_p}{\longrightarrow}, \underset{L_q}{\longrightarrow}, O|\mathsf{writes}]\end{aligned}$$

Sequential consistency models a memory system that behaves equivalently to a serial memory that only allows one processor to access the memory at a time. This forms a total order on the operations on the memory. Sequentially consistent memories can perform various optimizations as long as the observable behaviour "looks like" that of a serial memory.

The following example will demonstrate the notation introduced earlier. Consider the following multi-process:

$$P = \begin{cases} p: & x \leftarrow 5 \\ q: & \mathtt{read}_q(x) \end{cases}$$

Assuming x is a non-negative integer variable, the computations generated by the program $(P, J = \{x\})$ are:

$$\mathcal{C}\,(P, J) = \left\{ \begin{bmatrix} p: & \mathtt{write}(x,5) \\ q: & \underline{\mathtt{read}(x)} \\ & 0 \end{bmatrix}, \begin{bmatrix} p: & \mathtt{write}(x,5) \\ q: & \underline{\mathtt{read}(x)} \\ & 1 \end{bmatrix}, \begin{bmatrix} p: & \mathtt{write}(x,5) \\ q: & \underline{\mathtt{read}(x)} \\ & 2 \end{bmatrix}, \cdots \right\}$$

The system $(P, J, \mathsf{SequentialConsistency})$, will filter the computations of (P, J) for those that satisfy SequentialConsistency.

Consider the following computation:

$$C = \left[\begin{array}{ll} p: & \texttt{write}(x,5) \\ q: & \dfrac{\texttt{read}(x)}{5} \end{array}\right]$$

Consider the following total orders:

$$(O_C|p \cup O_C|\mathsf{writes}, \underset{L_p}{\longrightarrow}) = \Big(\texttt{write}(x,5)\Big)$$
$$(O_C|q \cup O_C|\mathsf{writes}, \underset{L_q}{\longrightarrow}) = \Big(\texttt{write}(x,5) \underset{L_q}{\longrightarrow} \frac{\texttt{read}(x)}{5}\Big)$$

Observe that they are witness orders to $\mathsf{SequentialConsistency}[C]$. These witness orders are often called *views*, since they represent the order that each process "sees" the operations in. Sequential consistency requires that the views of the processes all agree on the order of writes. There is only one other computation of the program that satisfies SequentialConsistency, therefore:

$$\mathcal{C}\left(P, J, \mathsf{SequentialConsistency}\right) = \left\{ \left[\begin{array}{ll} p: & \texttt{write}(x,5) \\ q: & \dfrac{\texttt{read}(x)}{0} \end{array}\right], \left[\begin{array}{ll} p: & \texttt{write}(x,5) \\ q: & \dfrac{\texttt{read}(x)}{5} \end{array}\right] \right\}$$

If we restrict the conditions of SequentialConsistency to apply to only operations of the same variable, we get Coherence.

$$\begin{aligned} \mathsf{Coherence}[C] \overset{\text{def}}{=} \mathbf{[VTO]}\exists \langle\ & (O|p \cup O|\mathsf{writes}, \underset{L_p}{\longrightarrow}) \text{ valid total order} : p \in P \\ & : (\forall x \in J : \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O|p \cup O|\mathsf{writes}(\{x\})]) \ \rangle : \\ & \mathbf{[AGR]}\forall p, q \in P, x \in J : \mathsf{Agree}[\underset{L_p}{\longrightarrow}, \underset{L_q}{\longrightarrow}, O|\mathsf{writes}(\{x\})] \end{aligned}$$

Coherence is a common basic requirement for memory models, introduced early in the literature [25, 21]. It requires the different views of a variable to agree on the order of operations applied to it.

The Pipelined-RAM model introduced by Lipton and Sandberg [40] is defined as follows:

$$\mathsf{Pipelined\text{-}RAM}[C] \stackrel{\text{def}}{=} [\mathbf{VTO}]\exists \langle\, (O|p \cup O|\mathsf{writes}, \underset{L_p}{\longrightarrow}) \text{ valid total order} : p \in P$$
$$: \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O|p \cup O|\mathsf{writes}] \,\rangle$$

It is intended to model a replicated shared memory where write updates are broadcasted through FIFO channels. Each process sees the writes of another process in the order that they were issued, but there are no guarantees on how each process orders two operations from two different processes.

The formalization of Goodman's processor consistency presented by Ahamad *et. al.* [4] is presented below. It can be interpreted as requiring the witness orders $\langle \underset{L_p}{\longrightarrow} :: \rangle$ to satisfy the constraints of both Pipelined-RAM and Coherence. This is not the same as the conjunction of the models, since the conjunction would allow two different witness orders to satisfy each set of constraints separately.

$$\mathsf{PC\text{-}G}[C] \stackrel{\text{def}}{=} [\mathbf{VTO}]\exists \langle\, (O|p \cup O|\mathsf{writes}, \underset{L_p}{\longrightarrow}) \text{ valid total order} : p \in P$$
$$: \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O|p \cup O|\mathsf{writes}] \,\rangle :$$
$$[\mathbf{AGR}]\forall p, q \in P, x \in J : \mathsf{Agree}[\underset{L_p}{\longrightarrow}, \underset{L_q}{\longrightarrow}, O|\mathsf{writes}(\{x\})]$$

The next model we will see generalizes Pipelined-RAM, PC-G, and SequentialConsistency.

3.1.1 Partition Consistency platform

Partition Consistency is a class of platforms consisting of single-threaded processes that communicate only by shared read/write variables. Each platform in the Partition Consistency class is determined by a parameter K, that is a partition of some subset of the set of variables, J. This partition determines the memory consistency model of the system. We will assume that K takes the form:

$$K = \{S_1, S_2, \ldots, S_k\}$$

Definition 3.1.1. *Let C be a computation of a program (P,J), then*

$$\begin{aligned}\mathsf{PartitionConsistency}(K)[C] \stackrel{\text{def}}{=}\\ &[\mathbf{VTO}]\exists\ \langle\ (O|p \cup O|\mathsf{writes}, \underset{L_p}{\longrightarrow})\ \textit{valid total order} : p \in P\\ &\quad : \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O|p \cup O|\mathsf{writes}]\ \rangle :\\ &\quad [\mathbf{AGR}]\forall p, q \in P, i \in [1,k] : \mathsf{Agree}[\underset{L_p}{\longrightarrow}, \underset{L_q}{\longrightarrow}, O|\mathsf{writes}(S_i)]\end{aligned}$$

Let V_J denote the subset of the set of objects J that are read/write variables (omitting the subscript if J is clear). On the Partition Consistency platform, $V_J = J$. A variable is a *single-writer variable* if it can be written by only one process, otherwise it is a *multi-writer variable*. The multi-writer variable subset of V will be denoted by $V|\mathsf{multi\text{-}writers}$.

Note that the Pipelined-RAM model is captured by the partial order [**VTO**] constraints. The agreement property is determined by K. By strengthening/weakening this additional agreement, we can obtain Pipelined-RAM, SequentialConsistency, and PC-G as special cases:

- Pipelined-RAM is PartitionConsistency($\emptyset$);
- SequentialConsistency is PartitionConsistency($\{V\}$);

- PC-G is PartitionConsistency(G) where for each $v \in J, \{v\} \in G$.

The sets of the partition K are groups of variables that jointly have strong consistency. This gives a very flexible definition, which could have multiple uses. For example, if we have two independent multiprograms, that do not interact, running on separate processes, then we may group their variables individually, and maintain weaker consistency between them. Another example would be that in a specific hardware implementation, it may be a lot cheaper or faster to only maintain strong consistency within certain bounds, such as a memory page. A final example is that some programming languages, such as Java [5], have a very weak memory consistency model by default. To compensate, it can be manually specified that some variables have stronger consistency requirements than others. These variables would be placed in the same group, this can be seen in Chapter 7.

If $\{x\} \in K$ and x is a single writer variable, then the agreement property holds immediately (write operations on x are totally ordered by program order). Thus the size of K can be reduced while maintaining PartitionConsistency(K). Though the specifications remain the same, implementations must spend extra resources to maintain the consistency of each set in K. This observation allows us to reduce partition maintenance overhead. This motivates the definition of two new models:

- WeakSC $\stackrel{\text{def}}{=}$ PartitionConsistency($\{V|$multi-writers$\}$)
- WeakPC-G $\stackrel{\text{def}}{=}$ PartitionConsistency(G) where for each $v \in V|$multi-writers$, \{v\} \in G$.

We can see from the previous discussion that WeakPC-G is equivalent to PC-G. WeakSC however, is a new model which is stronger than PC-G but weaker than SequentialConsistency.

3.1.2 Threaded Network platform

The Threaded Network platform is a system consisting of multithreaded processes, which we call a network. It is the lowest level platform in our presentation and it corresponds to the implementation platform used for the performance experiments.

The objects of the Threaded Network platform are variables that are only shared locally between threads of the same process and the single network object that is globally shared. The network object has two operations, `send`(s, d, m) (source, destination, message) and `recv`(), with the property that every message that has been received must have been sent, and that each message may be sent at most once and received at most once (we assume some mechanism for making each message unique). The `recv`() returns the arguments of its corresponding send, so $\frac{\texttt{recv}()}{s, d, m}$ corresponds to `send`(s, d, m). Since we use the memory consistency model to capture most of the properties of the network object we define all sequences of `send`() and `recv`() operations to be valid for the network object.

Our timestamp based implementation (Section 5.3) will further require priority queue objects that are locally shared on each node, specifically min priority queues. The priority queue object has the operations `enqueue`(), `extractmin`(), and `peekmin`(). These assume that the items queued and returned are totally ordered. For our uses, the only items that are queued are totally ordered, timestamped messages. The valid sequences of operations are those in which each `peekmin`() or `extractmin`() returns the smallest item that was previously enqueued that was not previously returned by a `extractmin`().

Define the following relations on a computation C that satisfies the above object constraints:

$$x \underset{\text{MessageOrder}_C}{\longrightarrow} y \stackrel{\text{def}}{=} x, y \in O_C \wedge x = \texttt{send}(s, d, m) \ \wedge y = \frac{\texttt{recv}()}{s, d, m} \tag{3.1.1}$$

$$x \underset{\text{WritesInto}_C}{\longrightarrow} y \stackrel{\text{def}}{=} x, y \in O_C \wedge x = \texttt{write}(w, z) \wedge y = \frac{\texttt{read}(w)}{z} \tag{3.1.2}$$

$$x \underset{\text{FifoChannel}_C}{\longrightarrow} y \stackrel{\text{def}}{=} x, y \in O_C \wedge \frac{\texttt{recv}()}{s, d, m} \wedge y = \frac{\texttt{recv}()}{s, d, m'} \wedge \texttt{send}(s, d, m) \underset{\text{prog}}{\longrightarrow} \texttt{send}(s, d, m') \tag{3.1.3}$$

$$\underset{\text{HappensBefore}_C}{\longrightarrow} \stackrel{\text{def}}{=} (\underset{\text{MessageOrder}}{\longrightarrow} \cup \underset{\text{prog}}{\longrightarrow} \cup \underset{\text{WritesInto}}{\longrightarrow} \cup \underset{\widehat{\text{FifoChannel}}}{\longrightarrow})^+ \tag{3.1.4}$$

The subscript is dropped when the computation C can be assumed from the context. $\underset{\text{HappensBefore}_C}{\longrightarrow}$ is defined always on O_C. Definitions may ignore certain pairs in the relation, but only one such relation is created for each computation. Given these relations, define the ThreadedNetwork model:

Definition 3.1.2. *Let C be a computation of a program (P, J) then:*

$$\mathsf{ThreadedNetwork}[C] \stackrel{\text{def}}{=}$$
$$[\mathbf{VTO}] \exists \ \langle \ (O|p, \underset{L_p}{\longrightarrow}) \ \textit{valid total order} \ : p \in P : \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{HappensBefore}}{\longrightarrow}, O|p] \ \rangle$$

This model represents a message passing network of multithreaded nodes. Each node has a shared memory that is sequentially consistent, so we require a total view order of the operations of each node that satisfies the sequential specification. Next we require a causal order, $\underset{\text{HappensBefore}}{\longrightarrow}$. This ensures several properties:

- Causality between messages: a message must be received after it is sent
- Causality between writes and reads: a value must be written before it is read
- FIFO ordering between messages: two messages sent in program order must be received in that order

- Combined causality between messages, reads, writes and FIFO order.

The last point is needed for our token algorithm in particular. Shared variable handshake is used to synchronize the main thread with a token thread and the token thread uses message passing to synchronize with the other nodes. The combined causal order is needed to maintain correctness here.

By combining the requirements of the causal order with the local views, we obtain a model of a message passing network.

Note that the $\underset{\text{HappensBefore}}{\longrightarrow}$ relation referred to in the definition is $\underset{\text{HappensBefore}_C}{\longrightarrow}$. Observe also that if $\exists o \in O : o \underset{\text{HappensBefore}}{\longrightarrow} o$ then there can be no total order extension of $\underset{\text{HappensBefore}}{\longrightarrow}$ containing o therefore:

$$\mathsf{ThreadedNetwork}[C] \implies (O, \underset{\text{HappensBefore}}{\longrightarrow}) \text{ partial order} \tag{3.1.5}$$

We may build a Partition Consistency platform directly on this level, however, we obtain cleaner proofs and better abstraction by introducing a middle layer. This middle layer separates the fact that processes broadcast write updates and apply them locally from how the broadcasting is managed.

3.1.3 Partial-order-broadcast cluster platform

Similar to the Threaded Network platform, the Partial-order-broadcast cluster platform has variables that are shared locally between threads of the same process. The key difference is the globally shared broadcast object that replaces the point-to-point network object. The operations of the broadcast object are `bcast`() and `deliver`(). The object has the property that every delivered update must have been `bcast`(), each update may be `bcast`() at most once, and each update may be delivered at most once per process. The valid sequences of the broadcast object are all those such that the `bcast`() of an update does not come after its corresponding `deliver`(), with further

constraints left to the memory consistency model.

One important feature of this platform is that `bcast`()ed updates may be labeled by labels in the set L determined by the platform. This labeling serves to force certain update orderings. Let has-label(m) be a boolean indicating whether or not m has a label and let label(m) denote the label of an update. label(m) is undefined for unlabeled updates.

Define a relation to capture the FIFO ordering of `bcast`()ed updates:

$$\underset{delorder}{\longrightarrow} \stackrel{\text{def}}{=} \left\{ \left(\frac{\texttt{deliver()}}{m_1}, \frac{\texttt{deliver()}}{m_2} \right) : \texttt{bcast}(m_1) \underset{\text{prog}}{\longrightarrow} \texttt{bcast}(m_2) \right\}$$

Given this relation, define the Partial-order-broadcast cluster memory consistency model:

Definition 3.1.3. *Let C be a computation of a program (P, J)*

$$\begin{aligned}
&\mathsf{POB\text{-}Cluster}(L)[C] \stackrel{\text{def}}{=} \\
&[\mathbf{VTO}] \exists \langle\ (O|p, \underset{L_p}{\longrightarrow}) \textit{valid total orders} : p \in P : \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow} \cup \underset{delorder}{\longrightarrow}, O|p]\ \rangle : \\
&\qquad [\mathbf{AGR}] \forall p, q \in P : (\text{label}(m_1) = \text{label}(m_2) \\
&\qquad\qquad \wedge \frac{\texttt{deliver()}}{m_1} \underset{L_p}{\longrightarrow} \frac{\texttt{deliver()}}{m_2} \\
&\qquad\qquad \wedge \frac{\texttt{deliver()}}{m_2} \in O|q \\
&\qquad) \implies \frac{\texttt{deliver()}}{m_1} \underset{L_q}{\longrightarrow} \frac{\texttt{deliver()}}{m_2}
\end{aligned}$$

The definition of this broadcast platform models a network of multithreaded nodes that communicate using a one-to-all broadcast operation and a corresponding deliver operation. The local view requirement is similar to that of the network platform. The definition uses an agreement property between delivers to enforce broadcast ordering properties. For a totally ordered broadcast, this agreement would be exact. For partially ordered broadcast, exact agreement is only enforced between messages of the

same label, with only FIFO enforced between all other messages. For the proofs of correctness, we only require a local causality, that a message can never be delivered before it is sent.

As noted, p is a multithreaded process, therefore $(O|p, \underset{\text{prog}}{\longrightarrow})$ is not necessarily a total order.

Observe that the definition of $\mathsf{POB\text{-}Cluster}(L)$ enforces a similar agreement on the deliveries of messages with the same label as $\mathsf{PartitionConsistency}(K)$ enforces on writes to variables of the same set in K. We will soon see that the relationship between $\mathsf{POB\text{-}Cluster}(L)$ and $\mathsf{PartitionConsistency}(K)$ is a generalization of the relationship between totally ordered broadcast and sequential consistency.

3.2 Transformation structure

In this modeling framework, we implement platform A on platform B by transforming programs for one platform A to programs for platform B. This is different from the other frameworks discussed in our related work. Most of the frameworks discussed in the related works section directly compare processes to other processes or specifications.

To prove that a distributed shared memory implementation implements a platform specified by a memory consistency model in an automata based framework, we would start by assigning each operation a transition, or action. We would then prove that the behaviour of this automata, the sequences of actions it produces, matches the memory consistency model. An example of this type of proof is presented by Attiya and Welch [8]. This means that the specification is a memory consistency model that abstracts the implementation details of the system and does not assume strong global time, but the implementation is built on an automata model that does assume strong global time.

In contrast, in this work, we start out specifying a specification level platform by describing the objects this platform supports and its memory consistency model, and a

target level platform in a similar fashion. Since the underlying memory systems on both levels are specified, the only way to implement the specification level platform on the target level platform is to "compile" or transform specification level programs to target level programs. There may be multiple possible program transformations that are correct and these transformations may be quite complex. To use program transformations as our basis for implementation requires a formal definition of these transformations and when such transformations are correct.

3.2.1 Program transformations

In this thesis, program transformations, which we will see examples of shortly, will be specified by:

1. Specifying a replacement for each operation invocation with code

   τ (`oper`())
   1 ...
   2 ⋮

2. Specifying threads to be added to each process in the form:

$$\tau.thread(P) \stackrel{\text{def}}{=} \begin{cases} \widehat{p}.t & \dots \\ \vdots & \vdots \end{cases}$$

We will call (P, J) the *specification program* and it's transformation $(\tau(P), \tau(J))$ the *target program*.

Pipelined RAM example transformation: We begin with a transformation that directly implements PipelinedRAM on the Threaded Network platform.

The first step involves replacing each operation invocation with code. We replace the read to the shared memory with a read to the local replica.

PipelinedRAMtrans($\texttt{read}_p(x)$)
1 **return** $Memory[p].x$

The write operation is replaced by a write to the local replica, then a loop that notifies the other processes to update their replicas.

PipelinedRAMtrans($\texttt{write}_p(x, v)$)
1 WRITE($Memory[p].x, v$)
2 **for** $q \in P \setminus \{p\}$
3 **do** SEND($p, q, [x, v]$)

These write notifications must be processed and applied to the local replica at each process. We accomplish this by adding a thread to each process to handle this task. We will use the notation $\widehat{p}$ to denote the process that results from the transformation of p.

$$\text{PipelinedRAMtrans}.thread(P) \stackrel{\text{def}}{=} \begin{cases} \forall p \in P : \widehat{p}.d : & \textbf{while} \quad \text{TRUE} \\ & \quad \textbf{do} \quad [x, v] \leftarrow \text{RECV}(); \\ & \qquad \text{WRITE}(Memory[p].x, v); \end{cases}$$

The transformation specification is complete, so we can now apply it to an example specification process:

$$P = \begin{cases} p.m : & x \leftarrow 5 \\ q.m : & \texttt{read}_q(x) \end{cases}$$

Applying the transformation yields a new target multiprogram.

$$\text{PipelinedRAMtrans}(P) = \begin{cases} \widehat{p}.m : & \text{WRITE}(Memory[p].x, 5); \\ & \textbf{for } r \in P \setminus \{p\} \\ & \quad \textbf{do SEND}(p, r, [x, 5]); \\ \widehat{p}.d : & \textbf{while TRUE} \\ & \quad \textbf{do } [x, v] \leftarrow \text{RECV}(); \\ & \qquad \text{WRITE}(Memory[p].x, v); \\ \widehat{q}.m : & \text{READ}(x); \\ \widehat{q}.d : & \textbf{while TRUE} \\ & \quad \textbf{do } [x, v] \leftarrow \text{RECV}(); \\ & \qquad \text{WRITE}(Memory[q].x, v); \end{cases}$$

Observe that the operations in the original program were replaced by their corresponding subroutines, and the threads added to the multiprogram as a whole.

More advanced example transformation: This example transformation is more advanced. We will describe it in detail and prove its correctness in a later section.

As in the previous example, we begin by replacing the read and write operations.

SWFRtrans($\texttt{read}_p(x)$)
1 **return** *Memory*$[p].x$

SWFRtrans($\texttt{write}_p(x, v)$)
1 *writes-requested* $\leftarrow$ *writes-requested* $+1$
2 **if** $\exists S_i \in K : x \in S_i$
3 **then** BCAST$([x, v, p, i])$
4 **else** BCAST$([x, v, p])$
5 **while** *writes-processed* $<$ *writes-requested*
6 **do skip**

As in the previous example, we must add a thread to handle write notifications. In this case the thread is more complicated as we are implementing a more complicated platform.

$$\text{SWFRtrans}.\mathit{thread}(P) \stackrel{\text{def}}{=} \left\{ \forall p \in P : \widehat{p}.d : \quad \textbf{while } \text{TRUE } \{\text{ApplyWrite}()\} \right.$$

ApplyWrite()
1 *update* $\leftarrow$ DELIVER()
{observe that *update* has form $[x, v, source]$ or $[x, v, source, l]$}
2 **let** $[x, v, source, -] = update$
3 **then** *Memory*$[p].x \leftarrow v$
4 **if** $source = p$
5 **then** *writes-processed* $\leftarrow$ *writes-processed* $+1$

Recall the specification process:

$$P = \left\{ \begin{array}{ll} p.m: & x \leftarrow 5 \\ q.m: & \texttt{read}_q(x) \end{array} \right.$$

Applying the SWFRtrans transformation to the process P yields a target process:

$$\text{SWFRtrans}(P) = \begin{cases} \widehat{p}.m: & \textit{writes-requested} \leftarrow \textit{writes-requested} + 1; \\ & \text{BCAST}([x, v, p]);\ \text{WaitWritesComplete}() \\ \widehat{p}.d: & \textbf{while true}\ \{\text{ApplyWrite}()\} \\ \widehat{q}.m: & \text{READ}(x) \\ \widehat{q}.d: & \textbf{while true}\ \{\text{ApplyWrite}()\} \end{cases}$$

The transformation is very syntactic, which simplifies our reasoning.

3.2.2 Model of subroutines

Subroutines are similar to programs, except that they return values. The return value of the transformation of an operation is important when we compare computations of the transformed, target, program to the computations of the original, specification, program.

Consider an object x that has only one operation `countandmultiplybytwo`$(x, \textit{value})$. Its valid sequences are all those in which each completed operation returns a pair containing the number of previous `countandmultiplybytwo`() operations on x and $\textit{value} \times 2$. This example contains both an operation on a shared count, and a purely local calculation. We do not model local calculation in detail since they do not interfere with other threads and processes.

Consider an transformation τ that must implement this operation on some platform.

τ(`countandmultiplybytwo`$(x, \textit{value})$)
1 $\textit{counter}[x] \leftarrow \textit{counter}[x] + 1$
2 **return** $(\textit{counter}[x], \textit{value} \times 2)$

This defines a program, and we can naturally extend the notation $\mathcal{C}(P, J)$ to this case. For example, for a specific invocation, such as `countandmultiplybytwo`$(x, 5)$, we have:

$$
\begin{aligned}
&\mathcal{C}\left(\tau(\texttt{countandmultiplybytwo}(x,5))\right)\\
=&\mathcal{C}\left(counter[x] \leftarrow counter[x]+1; \textbf{return}\quad (counter[x], 5\times 2);\right)\\
=&\left\{
\begin{array}{l}
\left[\begin{array}{c}\textsc{read}(counter[x])\\ \hline 0\end{array}, \textsc{write}(counter[x],1), \begin{array}{c}\textsc{read}(counter[x])\\ \hline 1\end{array}\right],\\[2ex]
\left[\begin{array}{c}\textsc{read}(counter[x])\\ \hline 1\end{array}, \textsc{write}(counter[x],2), \begin{array}{c}\textsc{read}(counter[x])\\ \hline 2\end{array}\right], \ldots\\[2ex]
\end{array}
\right\}
\end{aligned}
$$

Since **return** is not an operation, it does not show up in the computation. In this example, we may associate the final READ() with the return operation, but the local computation *value* $\times 2$ is lost. The return value must be represented in order to verify the correctness of the implementation.

Let $\mathcal{C}^{\textbf{return}}\left(\text{subroutine}(x,y,z,\ldots)\right)$ denote the sets of pairs of computations of subroutine(x,y,z) and their return values. For example:

$$
\begin{aligned}
&\mathcal{C}^{\textbf{return}}\left(\tau(\texttt{countandmultiplybytwo}(x,5))\right)\\
=&\mathcal{C}^{\textbf{return}}\left(counter[x] \leftarrow counter[x]+1; \textbf{return}\ (counter[x], 5\times 2);\right)\\
=&\left\{
\begin{array}{l}
\Big(\left[\begin{array}{c}\textsc{read}(counter[x])\\ \hline 0\end{array}, \textsc{write}(counter[x],1), \begin{array}{c}\textsc{read}(counter[x])\\ \hline 1\end{array}\right], (1,10)\Big),\\[2ex]
\Big(\left[\begin{array}{c}\textsc{read}(counter[x])\\ \hline 1\end{array}, \textsc{write}(counter[x],2), \begin{array}{c}\textsc{read}(counter[x])\\ \hline 2\end{array}\right], (2,10)\Big), \ldots\\[2ex]
\end{array}
\right\}
\end{aligned}
$$

Observe that we did not provide cases where the read performed for the **return** returned a different value than was written, but these computations would also be generated.

3.2.3 Trace transformation

To prove that the target system implements the specification system, we must compare their computations. To compare the computations of the target system, $\mathcal{C}\left(\tau(P), \tau(J), \widehat{\mathsf{MC}}\right)$, and the specification system, $\mathcal{C}\left(P, J, \mathsf{MC}\right)$, we need a way to relate them.

The simple nature of our program transformations allows us to obtain the set of transformed program computations by transforming the set of target program computations. This induced computation transformation, which we will call *trace transformation*, gives us a direct way of relating computations of the transformed program to those of the original.

For a transformation τ, we define its corresponding trace transformation τ^*. To define τ^* we start from the smallest pieces, operations, and build up inductively.

Let τ be a program transformation. We first define τ^α which obtains all the computations obtained by transforming a single operation.

$$\tau^\alpha(o) \stackrel{\text{def}}{=} \begin{cases} \mathcal{C}\left(\tau\left(\texttt{oper}(x, y, \dots)\right)\right) & \text{if} \quad o = \texttt{oper}(x, y, \dots) \\ \left\{C : (C, r_C) \in \mathcal{C}^{\textbf{return}}\left(\tau(\texttt{oper}(x, y, \dots)\right) \wedge r_C = r\right\} & \text{if} \quad o = \dfrac{\texttt{oper}(x, y, \dots)}{r} \end{cases}$$

A key requirement is if $\texttt{oper}(x, y, \dots)$ returns values of a certain type, then $\tau(\texttt{oper}(x, y, \dots))$ may only return values of that type. This requirement ensures that no computations of $\tau(\texttt{oper}(x, y, \dots))$ are filtered out if they return values of a different type.

We then define τ^δ which produces all the computations obtained by transforming a sequence of operations. This takes all possible combinations of the τ^α transformations.

$$\tau^{\delta}(\delta) \stackrel{\text{def}}{=} \begin{cases} \{\epsilon\} & \text{if} \quad \delta = \epsilon \\ \left\{\widehat{s} + \widehat{t} : \widehat{s} \in \tau^{\alpha}(o), \widehat{t} \in \tau^{\delta}(\delta')\right\} & \text{if} \quad \delta = o \cdot \delta' \end{cases}$$

Where $\cdot$ is sequence prepend and $+$ is sequence concatenation.

Next we define $\tau^{\mathcal{C}}$, the operation transformation phase of a computation transformation. Since a computation is just a collection of individual computations (sequences), this function collects all the combinations of the individual computation transformations.

$$\tau^{\mathcal{C}}\left(\left[\begin{array}{ll} p.a: & \delta_{p.a} \\ p.b: & \delta_{p.b} \\ q.a: & \delta_{q.a} \\ \vdots & \vdots \end{array}\right]\right) \stackrel{\text{def}}{=} \left\{\left[\begin{array}{ll} \widehat{p}.a: & \widehat{\delta} \in \tau^{\delta}(\delta_{p.a}) \\ \vdots & \vdots \end{array}\right]\right\}$$

The last step is to add threads. For our transformations, the additional threads form their own multiprogram, so we can take their computation set.

$$\tau^{\|} \stackrel{\text{def}}{=} \mathcal{C}\left(\tau.thread(P), \tau(J)\right)$$

Then to combine the operation transformation and thread addition phases define

$$\tau^{*}(C) \stackrel{\text{def}}{=} \tau^{\|} \parallel \tau^{\mathcal{C}}(C)$$

Where $A \parallel B = \{C \parallel C' : C \in A \wedge C' \in B\}$ and $\parallel$ on computations is just the merge of the two computations, the union of the two collections of individual computations (assuming thread names are disjoint).

Note that $\mathcal{C}\left(\tau(P), \tau(J)\right) = \tau^{*}(\mathcal{C}\left(P, J\right))$.

We can now define $\underset{\tau^{*-1}}{\models}$, which is the inverse of τ^{*} if τ^{*} is interpreted as a relation between sets of computations rather than a function from a computation to a set of

computations.

$$(\widehat{C} \underset{\tau^{*-1}}{\models} C) \stackrel{\text{def}}{=} \widehat{C} \in \tau^*(C) \tag{3.2.1}$$

3.2.4 An example of the SWFRtrans transformation

We clarify the formal definition of τ^* by applying it to the program and transformation we presented in section 3.2.1. Recall the example process and its transformation:

$$P = \begin{cases} p: & x \leftarrow 5 \\ q: & \texttt{read}_q(x) \end{cases}$$

$$\text{SWFRtrans}(P) = \begin{cases} \widehat{p}.m: & \textit{writes-requested} \leftarrow \textit{writes-requested} + 1; \\ & \text{BCAST}([x, v, p]);\ \text{WaitWritesComplete}() \\ \widehat{p}.d: & \textbf{while true } \{\text{ApplyWrite}()\} \\ \widehat{q}.m: & \text{READ}(x) \\ \widehat{q}.d: & \textbf{while true } \{\text{ApplyWrite}()\} \end{cases}$$

Computations of specification program: Recall from Section 3.1 that:

$$\mathcal{C}\left(P, \{x\}\right) = \left\{ \begin{bmatrix} p: & \texttt{write}(x,5) \\ q: & \underline{\texttt{read}(x)} \\ & 0 \end{bmatrix}, \begin{bmatrix} p: & \texttt{write}(x,5) \\ q: & \underline{\texttt{read}(x)} \\ & 1 \end{bmatrix}, \begin{bmatrix} p: & \texttt{write}(x,5) \\ q: & \underline{\texttt{read}(x)} \\ & 2 \end{bmatrix}, \cdots \right\}$$

Computations of specification system: The specification system

$(P, J, \mathsf{PartitionConsistency}(\{x\}))$, assuming 0 is the initial value of x, can only produce two computations:

$$\mathcal{C}\left(P, J, \mathsf{PartitionConsistency}(\{x\})\right) = \left\{ \begin{bmatrix} p: & \texttt{write}(x,5) \\ q: & \underline{\texttt{read}(x)} \\ & 0 \end{bmatrix}, \begin{bmatrix} p: & \texttt{write}(x,5) \\ q: & \underline{\texttt{read}(x)} \\ & 5 \end{bmatrix} \right\}$$

Trace transformation of operations: Rather than define this procedure formally, we illustrate the principle with an example. The procedure begins by defining the trace transformation of individual operations.

$\left[\begin{array}{ll} p: & \texttt{write}_p(x,5) \\ q: & \dfrac{\texttt{read}_q(x)}{5} \end{array}\right]$ has two operations, which we will transform.

We first transform $\texttt{write}_p(x,5)$:

$$\text{SWFRtrans}(\texttt{write}_p(x,5)) = \begin{cases} \textit{writes-requested} \leftarrow \textit{writes-requested} + 1; \\ \text{BCAST}([x,5,p]); \text{WaitWritesComplete}(); \end{cases}$$

This program generates an infinite set of sequences, here are a few:

$$\delta_1^w \stackrel{\text{def}}{=} \left[\begin{array}{l} \dfrac{\text{READ}(\textit{writes-requested})}{0}, \text{WRITE}(\textit{writes-requested}, 1), \\ \text{BCAST}([x,5,p]), \dfrac{\text{READ}(\textit{writes-processed})}{1}, \dfrac{\text{READ}(\textit{writes-requested})}{1} \end{array}\right]$$

$$\delta_2^w \stackrel{\text{def}}{=} \left[\begin{array}{l} \dfrac{\text{READ}(\textit{writes-requested})}{42}, \text{WRITE}(\textit{writes-requested}, 43), \\ \text{BCAST}([x,5,p]), \dfrac{\text{READ}(\textit{writes-processed})}{1}, \dfrac{\text{READ}(\textit{writes-requested})}{1} \end{array}\right]$$

$$\delta_3^w \stackrel{\text{def}}{=} \left[\begin{array}{l} \dfrac{\text{READ}(\textit{writes-requested})}{0}, \text{WRITE}(\textit{writes-requested}, 1), \\ \text{BCAST}([x,5,p]), \dfrac{\text{READ}(\textit{writes-processed})}{0}, \dfrac{\text{READ}(\textit{writes-requested})}{1}, \\ \textbf{skip}, \dfrac{\text{READ}(\textit{writes-processed})}{1}, \dfrac{\text{READ}(\textit{writes-requested})}{1} \end{array}\right]$$

skip is not an actual operation, but is added to indicate the action of the WaitWritesComplete while loop.

The trace transformation of the operation takes the form $\text{SWFRtrans}^{\alpha}(\texttt{write}_p(x,5)) = \{\delta_1^w, \delta_2^w, \delta_3^w, \dots\}$. It is the set of individual computations produced by the transformed operation's subroutine.

When the transformed operation has a response, we first generate the set of

computations using the arguments, then filter for the ones that return the correct value. The return statement exits the computation and ensures that this property holds, but does not show up as an operation.

We next obtain the trace transformation of $\frac{\texttt{read}_q(x)}{5}$. First observe that:

$$\mathcal{C}^{\textbf{return}}\left(\text{SWFRtrans}(\texttt{read}(x))\right) = \left\{\left(\left[\begin{array}{c}\frac{\text{READ}(Memory[q].x)}{v}\end{array}\right], v\right) : v \in \mathbb{N}\right\}$$

Then we can obtain the trace transformation by expanding the definition.

$$\begin{aligned}\text{SWFRtrans}^{\alpha}(\frac{\texttt{read}_q(x)}{5}) &= \left\{C : (C, r_C) \in \mathcal{C}^{\textbf{return}}\left(\text{SWFRtrans}(\texttt{read}(x))\right) \wedge r_C = 5\right\} \\ &= \left\{\delta_1^r = \left[\begin{array}{c}\frac{\text{READ}(Memory[q].x)}{5}\end{array}\right]\right\}\end{aligned}$$

Trace transformation of computations: We have already seen the trace transformation of an operation. To form the trace transformation of an entire computation, we first take all possible combinations of the trace transformations of its operations.

For example, if we wish to obtain the trace transformation the following computation:

$$C = \left[\begin{array}{ll} p: & \texttt{write}(x, 5) \\ q: & \frac{\texttt{read}(x)}{5}\end{array}\right]$$

The first form of the trace transformation will produce:

$$\text{SWFRtrans}^{\mathcal{C}}(C) = \left\{\left[\begin{array}{ll}\widehat{p}: & \delta_1^w \\ \widehat{q}: & \delta_1^r\end{array}\right], \left[\begin{array}{ll}\widehat{p}: & \delta_2^w \\ \widehat{q}: & \delta_1^r\end{array}\right], \left[\begin{array}{ll}\widehat{p}: & \delta_3^w \\ \widehat{q}: & \delta_1^r\end{array}\right], \ldots\right\}$$

Thread addition transformation: The trace transformation is completed by adding threads. The transformation so far cannot produce a consistent computation. None of the operation transformation code performs writes to the local replica. As mentioned before, our implementation uses the delivery thread to apply local and remote write updates.

The pseudocode for threads again specifies a set of sequences, we then take all possible combinations of the sequences. This set of computations is the trace transformation of the original computation.

$$\text{SWFRtrans}\,.thread(P) = \begin{cases} \widehat{p}.d: & \textbf{while}\ \text{TRUE}\ \{\text{ApplyWrite}();\} \\ \widehat{q}.d: & \textbf{while}\ \text{TRUE}\ \{\text{ApplyWrite}();\} \end{cases}$$

Computations of this are:

$$\text{SWFRtrans}^{\|} = \mathcal{C}\left(\text{SWFRtrans}\,.thread(P)\right)$$

A possible computation of $\widehat{p}.d$ is:

$$\delta_1^{pt1} \stackrel{\text{def}}{=} \left[\begin{array}{l} \dfrac{\text{DELIVER}()}{[x,5,p]}, \text{WRITE}(Memory[p].x, 5), \{[x,5,p].source = p\}, \dfrac{\text{READ}(wp)}{0}, \\ \text{WRITE}(wp, 1) \end{array} \right] \tag{3.2.2}$$

The comment in braces indicates that the comparison was made to select an if branch. These do not appear as operations in the computation.

A possible computation of $\widehat{q}.d$ is:

$$\delta_1^{qt1} \stackrel{\text{def}}{=} \left[\frac{\text{DELIVER}()}{[x,5,p]}, \text{WRITE}(Memory[q].x, 5), \{[x,5,p].source \neq q\} \right] \tag{3.2.3}$$

In this case the delivery thread $\widehat{q}.d$ did not increment *writes-processed* since the

message was not local ($p \neq q$).

This gives us some possible computations in SWFRtrans$^{\|}$:

$$\text{SWFRtrans}^{\|} = \mathcal{C}\left(\text{SWFRtrans}\,.thread(P)\right) = \left\{ \begin{bmatrix} \widehat{p}.d: & \delta_1^{pt1} \\ \widehat{q}.d: & \delta_1^{qt1} \end{bmatrix}, \begin{bmatrix} \widehat{p}.d: & \delta_1^{pt1} \\ \widehat{q}.d: & \end{bmatrix}, \begin{bmatrix} \widehat{p}.d: & \\ \widehat{q}.d: & \delta_1^{qt1} \end{bmatrix}, \ldots \right\}$$

Computations of the target program: The full trace transformation, SWFRtrans* is then:

$$\begin{aligned}
&\text{SWFRtrans}^{*}(C) \\
=\ &\\
&\text{SWFRtrans}^{\mathcal{C}}(C) \parallel \text{SWFRtrans}^{\|}(C) \\
=\ &\\
&\left\{ \begin{bmatrix} \widehat{p}: & \delta_1^{w} \\ \widehat{q}: & \delta_1^{r} \end{bmatrix}, \begin{bmatrix} \widehat{p}: & \delta_2^{w} \\ \widehat{q}: & \delta_1^{r} \end{bmatrix}, \begin{bmatrix} \widehat{p}: & \delta_3^{w} \\ \widehat{q}: & \delta_1^{r} \end{bmatrix}, \ldots \right\} \\
&\parallel \left\{ \begin{bmatrix} \widehat{p}.d: & \delta_1^{pt1} \\ \widehat{q}.d: & \delta_1^{qt1} \end{bmatrix}, \begin{bmatrix} \widehat{p}.d: & \delta_1^{qt1} \\ \widehat{q}.d: & \end{bmatrix}, \begin{bmatrix} \widehat{p}.d: & \\ \widehat{q}.d: & \delta_1^{qt1} \end{bmatrix}, \ldots \right\} \\
=\ &\\
&\left\{ \begin{bmatrix} \widehat{p}: & \delta_1^{w} \\ \widehat{p}.d: & \delta_1^{pt1} \\ \widehat{q}: & \delta_1^{r} \\ \widehat{q}.d: & \end{bmatrix}, \begin{bmatrix} \widehat{p}: & \delta_1^{w} \\ \widehat{p}.d: & \delta_1^{pt1} \\ \widehat{q}: & \delta_1^{r} \\ \widehat{q}.d: & \delta_1^{qt1} \end{bmatrix}, \ldots \right\}
\end{aligned}$$

We know that these computations are computations of the target program by the following fact noted in Section 3.2.3 :

$$\mathcal{C}\left(\tau(P), \tau(J)\right) = \tau^*(\mathcal{C}\left(P, J\right))$$

Computations of the target system: Take the first computation in our example:

$$\widehat{C_1}$$

$$=$$

$$\begin{bmatrix} \widehat{p}: & \delta_1^w \\ \widehat{p}.d: & \delta_1^{pt1} \\ q: & \delta_1^r \\ \widehat{q}.d: & \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} \widehat{p}.m: & \frac{\text{READ}(\textit{writes-requested})}{0}, \text{WRITE}(\textit{writes-requested}, 1), \text{BCAST}([x, 5, p]), \\ & \{[x, 5, p].\textit{source} = p\}, \frac{\text{READ}(\textit{writes-processed})}{1}, \frac{\text{READ}(\textit{writes-requested})}{1} \\ \widehat{p}.d: & \frac{\text{DELIVER}()}{[x, 5, p]}, \text{WRITE}(Memory[p].x, 5), \frac{\text{READ}(wp)}{0}, \text{WRITE}(wp, 1) \\ \widehat{q}.m: & \frac{\text{READ}(Memory[q].x)}{5} \\ \widehat{q}.d: & \end{bmatrix}$$

Notice in this computation that $\widehat{q}.m$ must read $Memory[q].x$ and return 5, but there is no such operation in $O|q$. Therefore $\widehat{C_1}$ does not satisfy $\mathsf{POB\text{-}Cluster}(L(K))$. The second computation in our example does provide this operation:

$$\widehat{C_2}$$

$$=$$

$$\begin{bmatrix} \widehat{p}: & \delta_1^w \\ \widehat{p}.d: & \delta_1^{qt1} \\ \widehat{q}: & \delta_1^r \\ \widehat{q}.d: & \delta_1^{qt1} \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} \widehat{p}.m: & \frac{\text{READ}(\textit{writes-requested})}{0}, \text{WRITE}(\textit{writes-requested}, 1), \text{BCAST}([x,5,p]), \\ & \{[x,5,p].source = p\}, \frac{\text{READ}(\textit{writes-processed})}{1}, \frac{\text{READ}(\textit{writes-requested})}{1} \\ \widehat{p}.h: & \frac{\text{DELIVER}()}{[x,5,p]}, \text{WRITE}(Memory[p].x, 5), \frac{\text{READ}(wp)}{0}, \text{WRITE}(wp, 1) \\ \widehat{q}.m: & \frac{\text{READ}(Memory[q].x)}{5} \\ \widehat{q}.h: & \frac{\text{DELIVER}()}{[x,5,p]}, \text{WRITE}(Memory[q].x, 5) \end{bmatrix}$$

The reader may verify that $\mathsf{POB\text{-}Cluster}(L(K))[\widehat{C_2}]$ is satisfied.

To prove the correctness of a transformation, we use the relationship between specification program and target program computations defined by trace transformation to show a similarity between the computations of their systems.

3.2.5 Interpretations and implementations

As noted previously, $\mathcal{C}(P, J)$ denotes the set of computations generated by a multiprogram (P, J). This generation is unrestricted, meaning that all computations are generated subject only to typing constraints of the objects. No memory consistency model is considered when generating this set.

Interpretations are relations between computations of systems. A certain class of

interpretations *consistent interpretations* will form the basis of our proof strategy.

Definition 3.2.1. *A relation* $\underset{R}{\models}$ *from* $\mathcal{C}(\widehat{P},\widehat{J})$ *to* $\mathcal{C}(P,J)$ *is called an* interpretation. *In the context of such a relation* (P,J) *is the* specified multiprogram *and* $(\widehat{P},\widehat{J})$ *is the* target multiprogram.

For readability, operations of the *specified multiprogram* will be denoted `oper`(), and operations of the *target multiprogram* will be denoted OPER().

Definition 3.2.2. *Let* (P,J,MC), $(\widehat{P},\widehat{J},\widehat{\mathsf{MC}})$ *be systems. Let* $\underset{R}{\models}$ *be an interpretation from the* target program $(\widehat{P},\widehat{J})$ *to the* specification program (P,J). *Then we say that* $\underset{R}{\models}$ *is a* consistent interpretation *from the* target system $(\widehat{P},\widehat{J},\widehat{\mathsf{MC}})$ *to the specification system* (P,J,MC) *if*

$$\forall C \in \mathcal{C}(P,J), \widehat{C} \in \mathcal{C}\left(\widehat{P},\widehat{J},\widehat{\mathsf{MC}}\right) : (\widehat{C} \underset{R}{\models} C) \implies C \in \mathcal{C}(P,J,\mathsf{MC})$$

Systems will be implemented by transforming their programs. To prove the correctness of these transformations, we will need to show that the interpretation induced by the transformation is a consistent one.

How do we define the interpretation of the computation of a transformed program? Consider the way that we obtain computations of the transformed program. We may transform the program directly then take its computations $\mathcal{C}(\tau(P),\tau(J))$. However, this gives us no way of relating the two sets of computations.

For a transformation τ, we use the relation induced by its trace transformation, $\underset{\tau^{*-1}}{\models}$, as an interpretation to relate the computations of the target program to computations of the specification program. A transformation is then an implementation if the interpretation $\underset{\tau^{*-1}}{\models}$ is consistent. Formally:

Definition 3.2.3. *Let* (P,J,MC) *be a system. Then* τ implements (P,J,MC) *system on a system* $(\tau(P),\tau(J),\widehat{\mathsf{MC}})$ *if* $\underset{\tau^{*-1}}{\models}$ *is a consistent interpretation.*

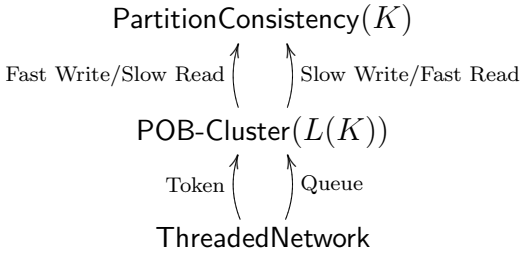
Note that this definition does not include any progress guarantees (discussed in Chapter 6). We only prove safety, also known as partial correctness, of implementations.

Definition 3.2.4. *Let* (J, MC), $(\widehat{J}, \widehat{\mathsf{MC}})$ *be platforms, and let* τ *be a program transformation. If for every program* (P, J), τ *implements* (P, J, MC) *on* $(\tau(P), \tau(J), \widehat{\mathsf{MC}})$ *then* τ platform implements (J, MC) *on* $(\widehat{J}, \widehat{\mathsf{MC}})$.

3.3 Proof structure

Our implementations proceed by first building the Partition Consistency platform on top of the Partial-order-broadcast cluster platform. Using Attiya and Welch's technique [7] we have two ways of doing this: a slow write/fast read version and a fast write/slow read version.

We then implement the Partial-order-broadcast cluster platform on the low level Threaded Network platform with two different transformations. One uses tokens to force agreement of message delivery between processes by mutual exclusion. The other uses priority queues and timestamping to provide the necessary agreement in message delivery. The following diagram illustrates the layered structure of our implementations and proofs.



3.3.1 Proof strategy

Definition 3.2.3 may be equivalently stated in terms of sets.

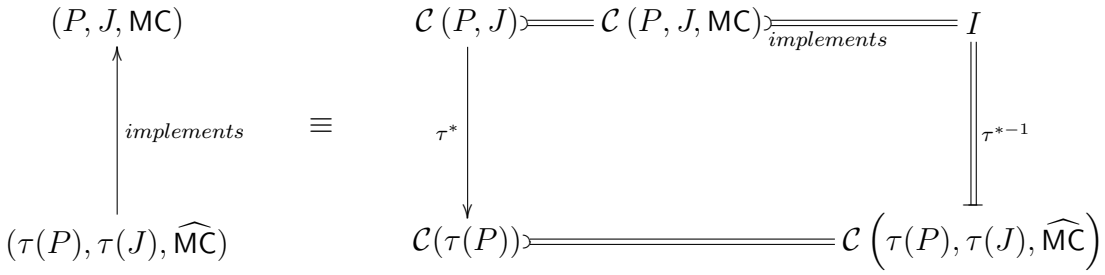
Definition 3.3.1. *Let I be the interpreted set of consistent target system computations:*

$$I = \{C : \exists \widehat{C} \in \mathcal{C}\left(\tau(P), \tau(J), \widehat{\mathsf{MC}}\right) : \widehat{C} \in \mathcal{C}\left(\tau(C)\right)\}$$

Then we say that τ implements (P, J, MC) *on* $(\tau(P), \tau(J), \widehat{\mathsf{MC}})$ *if*

$$I \subseteq \mathcal{C}\left(P, J, \mathsf{MC}\right)$$

Definition 3.3.1 is illustrated in the following diagram:



In these diagrams, we notate $A \subseteq B$ as $A \Subset B$ and equivalently $B \Supset A$.

We can prove the left hand side by proving the right hand side.

Outline of memory consistency proofs: The following is a general outline that our implementation proofs follow.

1. **Assume:**
   (a) Let $\widehat{C} \in \mathcal{C}(\tau(P), \tau(J), \widehat{\mathsf{MC}})$.
   (b) Let $C \in \mathcal{C}\left(P, J\right)$ such that $\widehat{C} \underset{\tau^{*-1}}{\models} C$
2. **Build:**
   (a) Choose some collection of witness orders $\widehat{A}$ that satisfy $\widehat{\mathsf{MC}}[\widehat{C}]$

(b) From these witness orders $\widehat{A}$, construct a collection of orders A to satisfy $\mathsf{MC}[C]$

3. **Verify:** Show that the constructed collection of orders A are witness orders to $\mathsf{MC}[C]$. This step generally forms the bulk of the proof and will usually be quite long.

3.3.2 Proof diagrams

Reasoning about memory consistency often requires many tedious but necessary chains of reasoning, often following partial orders operation by operation for several steps. A standard presentation of such a proof can be tiring to read and hard to grasp.

For this purpose we present these proofs in diagram form. This is inspired by similar reasoning that is common in category theory [11]. Each arrow in the diagram represents a boolean expression. Read the diagram as the conjunction of these expressions. Where possible, to follow the chain of reasoning, read the expressions, left to right, top to bottom. The basic building blocks are:

$a \longrightarrow b$ asserts that $a = b$

$a \xrightarrow{L} b$ asserts that $a \underset{L}{\longrightarrow} b$.

$a \xrightarrow{L} b$ asserts that $(a \underset{L}{\longrightarrow} b) \implies (c \underset{M}{\longrightarrow} d)$

$$\begin{array}{ccc} a & \xrightarrow{\quad L \quad} & b \\ \Big\| & & \Big\downarrow \\ c & \xrightarrow[\quad M \quad]{} & d \end{array}$$

$A \xrightarrow{L} B$ where A and B are sets asserts that $\forall a \in A, b \in B : a \underset{L}{\longrightarrow} b$.

$A \vdash \xrightarrow{L} B$ where A and B are sets asserts that $\exists b \in B : \forall a \in A : a \underset{L}{\longrightarrow} b$.

$A - \xrightarrow{L} B$ where A and B are sets asserts that $\exists a \in A : \forall b \in B : a \underset{L}{\longrightarrow} b$.

$A \subset B$ asserts that $A \subset B$.

$a - \underset{R}{-} b$ asserts that a is related to b by a relation R.

The notations $A \xrightarrow{L} B$, $A \vdash \xrightarrow{L} B$, and $A - \xrightarrow{L} B$ are similar to those used for Lamport's system executions [38].

When there are multiple edges between two nodes, this is read as the disjunction of the assertions. So $a \overset{E}{\underset{D}{\rightrightarrows}} b$ asserts that $a \underset{D}{\longrightarrow} b$ or $a \underset{E}{\longrightarrow} b$. In order to assert that $a \underset{D}{\longrightarrow} b$ and $a \underset{E}{\longrightarrow} b$ we would use:

$$\begin{array}{ccc} a & \underset{D}{\longrightarrow} & b \\ \| & & \| \\ a & \underset{E}{\longrightarrow} & b \end{array}$$

The direction of the edge will indicate which way it should be read. For example $a \underset{L}{\longrightarrow} b$ and $b \overset{L}{\longleftarrow} a$ are equivalent.

$\tau(o)$ is the set of operations that arose from the transformation of o. If it is a subroutine, then this is the subset of $\tau(o)$ that arose from the execution of that subroutine. If it is an operation name, then it is the unique operation of that type in $\tau(o)$. The notation mysub().$_n$OPER() is used to refer to the nth occurrence of an operation of the type OPER() in program order in an instance of a subroutine or operation transformations (which are specified by subroutines) mysub. The notation mysub().OPER() is similar but also asserts that only one instance of the operation type OPER() is in the instance. The meaning is similar for subroutine names such as mysub().$_n$ myothersub() or mysub(). myothersub(). This is combined with a numbering label mysub^n() that asserts that other nodes in the diagram with the same label n are equal and those with different labels are not equal.

Chapter 4

Implementing the Partition Consistency Platform using the Partial Order Broadcast Cluster Platform

4.1 SWFRtrans and FWSRtrans implementations

We wish to build a PartitionConsistency(K) shared memory on top of a Partial-order-broadcast cluster platform, POB-Cluster(L). The implementation we present is replicated; each process has a local, complete replica of the shared memory. When a process performs a write, it broadcasts a message to notify all the other processes to update their local replica. Recall that the parameter K, in PartitionConsistency(K) is a partition of a subset of the variables J. K specifies the sets of variables among which there is strong agreement. The message labels, L, are used to enforce this agreement between the operations applied to each of the partitions in K, by enforcing agreement on their corresponding notification messages. This makes L a function of K.

Figure 4.1 shows the specification Partition Consistency platform in a three process configuration, all communicating via shared variables.

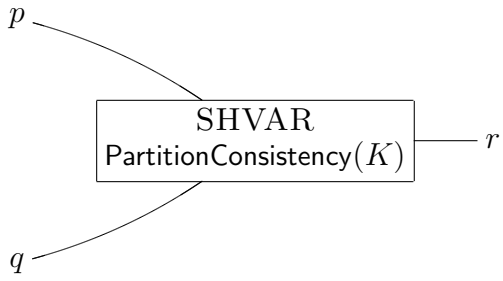


Figure 4.1: PartitionConsistency(K)

Figure 4.2 shows the target Partial-order-broadcast cluster platform, created by

transforming Figure 4.1. Each threaded process p in P only has a single thread $p.m$, which is transformed to the *main* thread $\widehat{p}.m$. The transformation also adds another thread, the *delivery* thread $\widehat{p}.d$, to each process $p \in P$. The joint computation of threads of the same process is Sequentially Consistent, but their interaction with threads of different processes satisfies a weaker consistency. Threads of different processes can only communicate with each other by using the single shared broadcast object.

The collection of globally shared variables is replaced by the single shared broadcast object. The implementation will simulate the global shared memory of the

PartitionConsistency(K) platform by replicating it on each of the local shared memories and using the properties of the broadcast object to maintain the required consistency specified by PartitionConsistency(K) between the replicas.

The replica of memory at process p is denoted $Memory[p]$. The replica of a specific variable x at process p is denoted $Memory[p].x$. Writes are implemented by broadcasting a corresponding update, then delivering that update to each process's local replica. An update corresponding to the operation `write`(x, v) by process p has the structure $[x, v, p]$ if unlabeled and $[x, v, p, l]$ if labelled. The read of a variable x is implemented by reading its corresponding local replica $Memory[p].x$.

The main thread $\widehat{p}.m$ is defined as a transformation of process p. That is, the read and write invocations are replaced by subroutines that implement their specification in the Partial-order-broadcast cluster layer. We use Attiya and Welch's technique [8] to create Slow Write and Fast Write versions of our main thread transformation. As discussed in Section 2.1, a fast operation is one does not incur any message delay.

The variables *writes-requested* and *writes-processed* are synchronization variables local to each process $\widehat{p}$ and shared between $\widehat{p}.m$ and $\widehat{p}.d$. They are used to track how many locally requested writes have been applied to the replica, which is important for maintaining program order.

Slow Write/Fast Read: The transformation of a read is straight forward, it simply

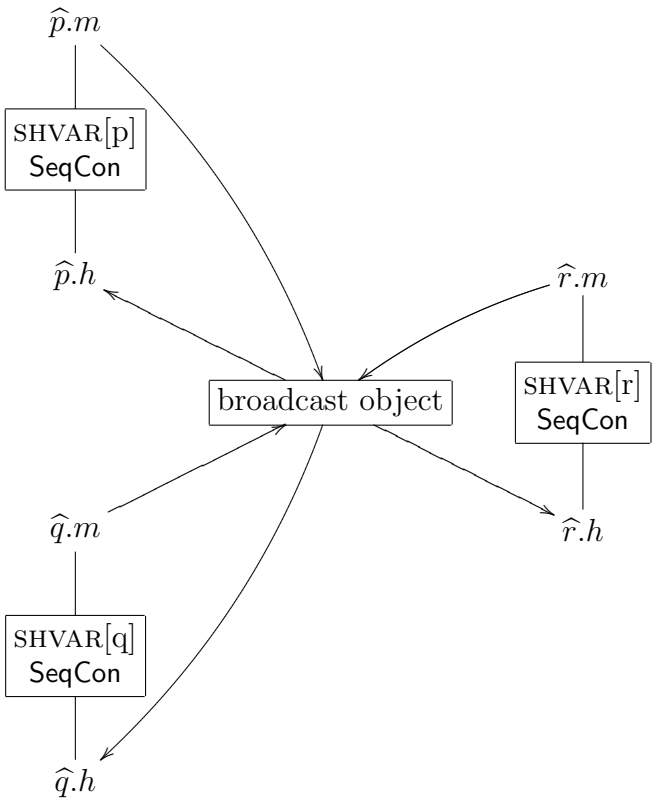


Figure 4.2: POB-Cluster(L(K))

returns the value stored in $\widehat{p}$'s local memory:

SWFRtrans($\texttt{read}_p(x)$)
1 **return** $Memory[p].x$

The transformation of a `write`() labels their corresponding update messages with the number of their corresponding partition K if needed. If the object is not in K, the transformation sends an unlabeled message.

SWFRtrans($\texttt{write}_p(x, v)$)
1 *writes-requested* $\leftarrow$ *writes-requested* $+1$
2 **if** $\exists S_i \in K : x \in S_i$
3 **then** BCAST($[x, v, p, i]$)
4 **else** BCAST($[x, v, p]$)
5 WaitWritesComplete()

The procedure WaitWritesComplete requires the main thread $\widehat{p}.m$ to wait until the delivery thread $\widehat{p}.d$ has applied all the writes previously broadcast by $\widehat{p}.m$. In a system

implemented with the SWFRtrans() transformation, each process may have at most one outstanding local write at any time, since every write must be applied locally before the subroutine completes. Every write contains a wait, making these writes "slow".

```
WaitWritesComplete()
1  while writes-processed < writes-requested
2        do skip
```

Fast Write/Slow Read: The Fast Write/Slow Read implementation, denoted

FWSRtrans, implements operations similarly to SWFRtrans(). The difference is that the "slow" WaitWritesComplete() call has been moved from the write transformation to the read transformation.

```
FWSRtrans(write_p(x, v))
1  writes-requested ← writes-requested +1
2  if ∃S_i ∈ K : x ∈ S_i
3     then BCAST([x, v, p, i])
4     else BCAST([x, v, p])
```

Now, the read implementation must wait until all writes that are issued locally are applied locally.

```
FWSRtrans(read_p(x))
1  WaitWritesComplete()
2  return Memory[p].x
```

This implementation pipelines writes. Pipelined writes can "complete" instantly, and do not have to wait for a network response. For example, if a process performs a sequence of two writes, $[\texttt{write}(x,1), \texttt{write}(x,2)]$, both writes may complete before either of them is applied to the local replica.

Common delivery threads : The delivery thread $\widehat{p}.d$ indefinitely applies received writes to $Memory[p]$. Its implementation is the same for both SWFRtrans and

FWSRtrans transformations.

$$\begin{matrix}\text{SWFRtrans}.thread(P)\\ \text{FWSRtrans}.thread(P)\end{matrix} \stackrel{\text{def}}{=} \left\{ \forall p \in P : \widehat{p}.d : \quad \textbf{while } \text{TRUE } \{\text{ApplyWrite}()\} \right.$$

The ApplyWrite subroutine handles applying delivered write updates and maintaining the synchronization variables.

```
ApplyWrite()
1   update ← DELIVER()
    {observe that update has form [x, v, source] or [x, v, source, l]}
2   let [x, v, source, −] = update
3       then Memory[p].x ← v
4            if source = p
5               then writes-processed ← writes-processed +1
```

Observe by examining the transformation that messages are only labelled with partition numbers, so $L(K) = \{i : S_i \in K\}$.

4.2 Correctness of SWFRtrans and FWSRtrans implementations

The proofs of FWSRtrans and SWFRtrans are very similar, they only differ in one step. This step can be treated generically, so we present one proof for both implementations. Let WRtrans refer to either implementation.

Let (P, J) be a program compatible with the PartitionConsistency(K). This means that (P, J) must be a single threaded program that only operates on read/write variables.

Theorem 4.2.1. *The system* $(\text{WRtrans}(P), \text{WRtrans}(J), \text{POB-Cluster}(L(K)))$ *implements* $(P, J, \text{PartitionConsistency}(K))$.

Begin Proof

We use the general outline of memory consistency proofs from Section 3.3.1.

Assume: Let $\widehat{C}$ be a computation in $\mathcal{C}\left(\text{WRtrans}(P), \text{WRtrans}(J), \textsf{POB-Cluster}(L(K))\right)$ and let C be a computation in $\mathcal{C}\left(P, J\right)$ such that $\widehat{C} \underset{\text{WRtrans}^{*-1}}{\models} C$. Let $\widehat{O}$ denote the set of operations $O_{\widehat{C}}$.

Build: We construct orders $\langle\ (O_C|p \cup O_C|\textsf{writes}, \underset{L_p}{\longrightarrow}) : p \in P :\ \rangle$ to satisfy the partial order and agreement constraints of $\textsf{PartitionConsistency}(K)$.

Let $\langle\ (O_{\widehat{C}}|\widehat{p}, \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow}) : p \in \text{WRtrans}(P) :\ \rangle$ be witness orders to $\textsf{POB-Cluster}(L(K))[\widehat{C}]$.

For build steps we will abuse our notation. For a total order $(O, \underset{X}{\longrightarrow})$, we denote its induced sequence by X. Construct $\langle\ (O_C|p \cup O_C|\textsf{writes}, \underset{L_p}{\longrightarrow}) : p \in P :\ \rangle$ as follows:

1. For each operation o on a "local replica" variable we associate it with a specification level operation by defining a relation $\underset{construct}{\sim}$

   (a) $\dfrac{\text{READ}(Memory[p].x)}{v}$ by the transformation, this operation must have come from the transformation of a specification level $\dfrac{\texttt{read}(x)}{v} \in O|p$. Associate it with this specification level operation by letting:

   $$\frac{\text{READ}(Memory[p].x)}{v} \underset{construct}{\sim} \frac{\texttt{read}(x)}{v}$$

   (b) $\text{WRITE}(Memory[p].x, v)$ must have a $o_d = \dfrac{\texttt{deliver}()}{[\text{WRITE}, x, v]}$ in the same ApplyWrite() call. This deliver operation o_d must have a corresponding $\texttt{bcast}(m)$, which can only have occurred in the transformation of some high level write WRtrans($\texttt{write}(x, v)$). Associate the original operation with this specification level one by letting:

   $$\text{WRITE}(Memory[p], x, v) \underset{construct}{\sim} \texttt{write}(x, v)$$

2. Build the sequence $Short(\widehat{L}_{\widehat{p}})$ from the sequence $\widehat{L}_{\widehat{p}}$ by removing all of the

operations of the broadcast object, and the variables *writes-processed* and *writes-requested*. This leaves only the operations on the "local replica" variables.

3. Build the sequence L_p as follows: every operation in $Short(\widehat{L}_{\widehat{p}})$ is a local memory read or write on the target so we replace them with their associated high level operations. This sequence induces the required total order.

Verify: We prove that the constructed orders $\underset{L_p}{\longrightarrow}$ are witnesses to $\mathsf{PartitionConsistency}(K)[C]$.

To show this, we directly prove each of the properties in Definition 3.1.1.

The constructed sequences $\langle\ L_p : p \in P :\ \rangle$ induce corresponding total orders $\langle\ (O|p \cup O|\mathsf{writes}, \underset{L_p}{\longrightarrow}) : p \in P :\ \rangle$. These total orders are valid by the way that they were constructed from the valid witness total orders $\langle\ \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow} : \widehat{p} \in \widehat{P}\ \ \rangle$. Removing all operations related to specific objects preserves validity when forming $\mathrm{Short}(\widehat{L}_{\widehat{p}})$. Replacing reads and writes with corresponding reads and writes also preserves validity.

The remaining $\mathsf{POB\text{-}Cluster}(L(K))$ constraint are satisfied owing to the following Lemmas:

Constraint	Lemma
$\mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\mathrm{prog}}{\longrightarrow}, O\|p \cup O\|\mathsf{writes}]$	Lemma 4.2.2
$\forall p, q \in P, i \in [1,k] : \mathsf{Agree}[\underset{L_p}{\longrightarrow}, \underset{L_q}{\longrightarrow}, O\|\mathsf{writes}(S_i)]$	Lemma 4.2.3

Therefore $\mathsf{PartitionConsistency}(K)[C]$ as required.

End Proof

Let $\widehat{C} \in \mathcal{C}(\mathrm{WRtrans}(P), \mathrm{WRtrans}(J), \mathsf{POB\text{-}Cluster}(L(K)))$. Let C be such that $\widehat{C} \underset{\mathrm{WRtrans}^{*-1}}{\models} C$. This matches the **Assume** step of Theorem 4.2.1.

Choose witness orders $\langle\ (O_{\widehat{C}}|\widehat{p}, \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow}) : p \in \mathrm{WRtrans}(P) :\ \rangle$ that satisfy $\mathsf{POB\text{-}Cluster}(L(K))[\widehat{C}]$. Construct $\langle\ \underset{L_p}{\longrightarrow} : p \in P :\ \rangle$ based on the chosen witness orders as in the **Build** step of Theorem 4.2.1.

Lemma 4.2.2. $\forall p \in P$: Extends$[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O|p \cup O|\text{writes}]$

Begin Proof

Let o_1, o_2 be operations in $O|p \cup O|\text{writes}$ so that $o_1 \underset{\text{prog}}{\longrightarrow} o_2$. Since they are ordered by program order, they must have been invoked by the same process,

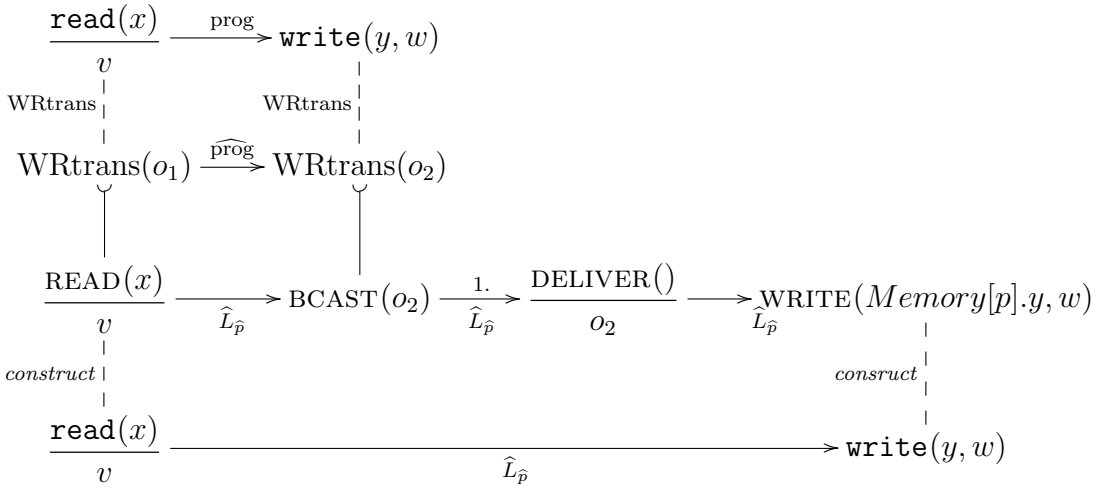
so $proc(o_1) = proc(o_2) = q$ for some q. This platform only has read/write variable objects, so there are four cases for o_1, o_2:

Case 1: read,read Then $o_1 = \dfrac{\texttt{read}(x)}{v}$ and $o_2 = \dfrac{\texttt{read}(y)}{w}$

We have $p = q$ and $\text{WRtrans}(o_1) \underset{\widehat{\text{prog}}}{\longrightarrow} \text{WRtrans}(o_2) \implies o_1 \underset{L_p}{\longrightarrow} o_2$ the reads are directly translated, so they remain in program order.

Case 2: read,write Then $o_1 = \dfrac{\texttt{read}(x)}{v}$ and $o_2 = \texttt{write}(y, w)$:

We have $p = q$ and $\text{WRtrans}(o_1) \underset{\widehat{\text{prog}}}{\longrightarrow} \text{WRtrans}(o_2)$ which implies



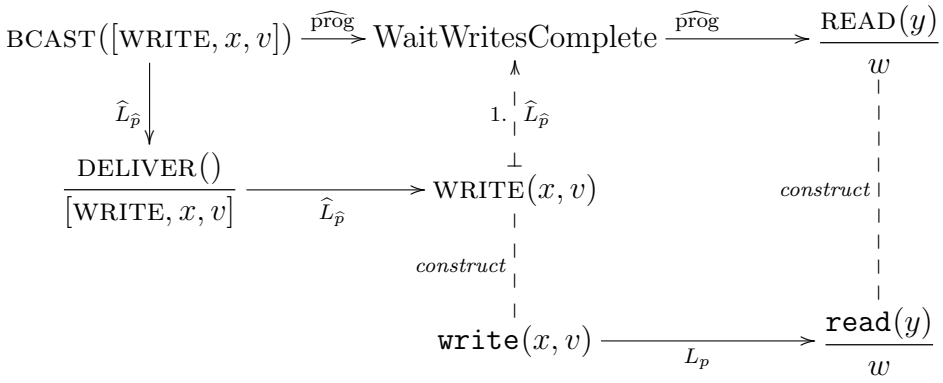
1. POB-Cluster$(L(K))$ that delivers must follow their corresponding broadcasts in the local order.

Case 3: write,write Then $o_1 = \texttt{write}(x, v)$ and $o_2 = \texttt{write}(y, w)$

Two writes in program order are also ordered by message order, so their delivers and associated writes must also follow this order.

Case 4: write,read $o_1 = \texttt{write}(x, v)$ and $o_2 = \dfrac{\texttt{read}(y)}{w}$

In both fast write and slow write algorithms there is a WaitWritesComplete call so that :



1. Property of WaitWritesComplete. All previously broadcasted writes must be delivered before it completes.

Thus in all cases we have $o_1 \xrightarrow[L_p]{} o_2$. Therefore $\xrightarrow[L_p]{}$ extends $\xrightarrow[\text{prog}]{}$.

End Proof

Lemma 4.2.3. $\forall p, q \in P, i \in [1, k]$: $\mathsf{Agree}[\xrightarrow[L_p]{}, \xrightarrow[L_q]{}, O|\mathsf{writes}(S_i)]$

Begin Proof

Let W_1, W_2 be write operations in some $O|\mathsf{writes}(S_i)$. Then the broadcasts of these writes in the transformation are $\text{BCAST}(msg_{w_1}) \in \text{WRtrans}(W_1)$

and $\text{BCAST}(msg_{w_2}) \in \text{WRtrans}(W_2)$. Since they share the same group, we can observe that $\text{label}(msg_{w_1}) = \text{label}(msg_{w_2})$.

For processes p and q there are target writes that are produced from the transformation of W_1 and W_2. Then let $w_{1p}, w_{1q}, w_{2p}, w_{2q}$ be these corresponding writes in $O_{\widehat{C}}$. Suppose WLOG that $w_{1p} \xrightarrow[\widehat{L}_{\hat{p}}]{} w_{2p}$, then $\frac{\text{DELIVER}()}{msg_{w_1}} \xrightarrow[\widehat{L}_{\hat{p}}]{} w_{1p} \xrightarrow[\widehat{L}_{\hat{p}}]{} \frac{\text{DELIVER}()}{msg_{w_2}} \xrightarrow[\widehat{L}_{\hat{p}}]{} w_{2p}$ which implies $\frac{\text{DELIVER}()}{msg_{w_1}} \xrightarrow[\widehat{L}_{\hat{q}}]{} w_{1q} \xrightarrow[\widehat{L}_{\hat{q}}]{} \frac{\text{DELIVER}()}{msg_{w_2}} \xrightarrow[\widehat{L}_{\hat{q}}]{} w_{2q}$ since $\mathsf{POB\text{-}Cluster}(L(K))$ requires that deliveries of messages with the same label must agree.

End Proof

Chapter 5

Implementing the Partial Order Broadcast Cluster Platform Using the Threaded Network Platform

The following lower level implementations build the Partial-order-broadcast cluster platform using the Threaded Network platform. The Threaded Network platform is similar to real world systems, with multithreaded nodes that communicate by message passing. This platform closely resembles the actual implementation platform, using the Message Passing Interface (MPI) for message passing between multithreaded nodes.

Unlike the FWSRtrans and SWFRtrans implementations of the previous section, the two implementations, TokenPOB and QueuePOB, use significantly different techniques to implement Partial-order-broadcast cluster.

In order to prove this transformation is an implementation, we need to show that it provides a consistent interpretation. This involves verifying that memory consistency constraints and operation restrictions hold. To facilitate this the object restrictions are incorporated into an equivalent definition of POB-Cluster(L). Most of the constraints are taken directly from the original POB-Cluster(L) definition and the operation restrictions of the broadcast object. The structure of the message agreement is significantly changed. The agreement on delivery order is changed from a pairwise agreement on deliveries between processes to a global agreement on a partial order of messages that is followed by all processes. This global partial order on messages, $(M, \underset{MO}{\longrightarrow})$, can be constructed from witness orders to the original definition by the constructing the order so that $(m_1 \underset{MO}{\longrightarrow} m_2)$ when $\forall p \in P : \texttt{bcast}(m_1) \underset{L_p}{\longrightarrow} \texttt{bcast}(m_2)$. Conversely, we can show that a witness $(M, \underset{MO}{\longrightarrow})$ enforces the agreement required by the original definition. The global partial order on messages, though more cumbersome, is more useful for the

following proofs as it can often be derived directly from the implementation structure.

Proving that the conditions of this definition hold will also show that the original memory consistency constraints and the object restrictions hold.

Definition 5.0.4.

$$
\begin{aligned}
&\mathsf{POB\text{-}Cluster}(L)[C] \stackrel{\text{def}}{=} \\
&\quad \exists\, (M, \underset{MO}{\longrightarrow})\ \textit{partial order}\ , \langle\ (O|p, \underset{L_p}{\longrightarrow}) : p \in P :\ \rangle : \\
&\qquad \left(\frac{\texttt{deliver}()}{m} \in O \implies m \in M \right) \\
&\qquad \wedge \left(\textit{a message is delivered only once per processor} \right) \\
&\qquad \wedge \left(\forall l \in L : (M|l, \underset{MO}{\longrightarrow})\ \textit{is total}\ \right) \\
&\qquad \wedge \left(\texttt{bcast}(m_1) \underset{\text{prog}}{\longrightarrow} \texttt{bcast}(m_2) \implies m_1 \underset{MO}{\longrightarrow} m_2 \right) \\
&\qquad \wedge \left(\texttt{bcast}(m), \frac{\texttt{deliver}()}{m} \in O|p \implies \texttt{bcast}(m) \underset{L_p}{\longrightarrow} \frac{\texttt{deliver}()}{m} \right) \\
&\qquad \wedge \left(m_1 \underset{MO}{\longrightarrow} m_2 \wedge \frac{\texttt{deliver}()}{m_2} \in O|p \implies \frac{\texttt{deliver}()}{m_1} \underset{L_p}{\longrightarrow} \frac{\texttt{deliver}()}{m_2} \right) \\
&\qquad \wedge \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O|p] \\
&\qquad \wedge \left((O|p, \underset{L_p}{\longrightarrow})\ \textit{is a valid total order}\ \right)
\end{aligned}
$$

5.1 TokenPOB implementation

The TokenPOB implementation relies on the FIFO property of the network to ensure FIFO delivery of broadcasted updates. A token for each label is added to maintain the agreement on the delivery of labeled updates.

Recall that we specify implementations by transforming operation invocations and adding threads. A big picture view of composing the TokenPOB transformation with the SWFRtrans transformation will start with a program involving only read/write variables:

$$P = \begin{cases} p.m: & \ldots, \texttt{write}(\ldots), \texttt{read}(\ldots) \ldots \\ q.m: & \ldots, \texttt{read}(\ldots), \ldots \end{cases}$$

The transformation will then replicate the read/write variables, and use the broadcast operation to notify other processes to update their replicas. A thread is added to each process to manage the notifications.

$$\text{SWFRtrans}(P) = \begin{cases} \widehat{p}.m: & \ldots, \text{BCAST}(\ldots), \ldots \\ \widehat{p}.d: & \ldots, \text{DELIVER}(), \ldots \\ \widehat{q}.m: & \ldots, \text{BCAST}(\ldots), \ldots \\ \widehat{q}.d: & \ldots, \text{DELIVER}(), \ldots \end{cases}$$

This transformed process is now transformed again, implementing the broadcast and receive operations using send and recv operations. The transformation uses a collection of tokens to synchronize processes, for each process, a thread is added for each token, to manage that token.

$$\text{TokenPOB}(L)(\text{SWFRtrans}(P)) = \begin{cases} \widehat{\widehat{p}}.m: & \ldots, \text{SEND}(), \ldots \\ \widehat{\widehat{p}}.d: & \ldots, \text{RECV}(), \ldots \\ \widehat{\widehat{p}}.t_1: & \ldots, \text{SEND}([\text{TOKEN}, \ldots]), \ldots \\ \vdots & \\ \widehat{\widehat{q}}.m: & \ldots, \text{SEND}(), \ldots \\ \widehat{\widehat{q}}.d: & \ldots, \text{RECV}(), \ldots \\ \widehat{\widehat{q}}.t_1: & \ldots, \text{SEND}([\text{TOKEN}, \ldots]), \ldots \\ \vdots & \end{cases}$$

It would be reasonable to have a single thread handle all the token maintenance messages. However, to simplify the proofs and the presentation, each token has one thread on each processor dedicated to its maintenance.

When a broadcast is performed, the thread may have to acquire a token. Tokens are acquired by using a handshake protocol to synchronize with the appropriate token thread. This handshake protocol is presented below:

```
ProtectedBlock(need,doorOpen,protectedAction)
1  need ← TRUE
2  while ¬ doorOpen skip
3  protectedAction()
4  doorOpen ← FALSE
5  need ← FALSE

Guard(need,doorOpen)
1  if need
2     then doorOpen ← TRUE
3          while need skip
```

The handshake algorithm has the property that every time Guard is called, it allows at most one call of ProtectedBlock to execute its protectedAction, and the Guard call will exit after the end of the ProtectedBlock. The transformation uses the

ProtectedBlock to acquire a token and perform any operations that require that token. The handshake algorithm will ensure that the token is not passed on until the ProtectedBlock completes.

Only broadcasts of labeled messages require a token. These must also wait for acknowledgments that their broadcasts were received to enforce total ordering. Unlabeled messages do not have these requirements and only need to be sent to all other processes. The transformation described below handles all of these details:

The processes in P are numbered starting at 0. Connect these processes in a ring as follows:

$$\text{next}(p) \stackrel{\text{def}}{=} (p+1) \textbf{ mod } |P|$$

This ring is used to pass the tokens. A token is created for each label $l \in L$, and for each token, a thread is created on each process to manage it.

$$\text{TokenPOB}(L).\mathit{thread}(P) \stackrel{\text{def}}{=}$$

$$\left\{ \begin{array}{ll} \forall p \in P, l \in L : & 1 \quad \textbf{if } p = 0 \\ \quad \widehat{p}.\,\text{TokenThread}_l : & 2 \quad\quad \textbf{then } \textsc{send}(p, \text{next}(p), [\textsc{token}, \textsc{bcastgrouptoken}_l]) \\ & 3 \quad \textbf{loop} \\ & 4 \quad\quad\quad \textbf{do } \text{PassToken}_p(l) \end{array} \right.$$

The PassToken subroutine groups one acquisition and release of a token. This grouping allows us to distinguish between separate token acquisitions. It calls the Guard subroutine from the handshake protocol each time it acquires a token to allow another thread to use the token. If the main routine is not waiting for the token, the Guard subroutine completes and the token is passed on.

Pattern matching is used to ensure token threads receive only the messages that are related to their token. $(\mathit{pattern}) \dashv \text{recv}()$ is pseudocode that blocks until a message matching *pattern* is received and stores the message in the appropriate pattern variables. For example, $(q, p, [\textsc{message}, m]) \dashv \text{recv}()$ blocks until a message with a

matching form can be received, parses out the relevant data from the message and stores it in the variables q and m.

PassToken$_p(l)$
1 $(q, p, [\text{TOKEN}, \text{BCASTGROUPTOKEN}_l]) \dashv \text{recv}()$
2 Guard(*need-token*$_l$, *door*$_l$)
3 $\text{SEND}(p, \text{next}(p), [\text{TOKEN}, \text{BCASTGROUPTOKEN}_l])$

The transformation of the bcast operation must acquire the appropriate token to broadcast a labeled message by synchronizing with its token thread. Unlabeled messages do not require a token to be broadcast.

TokenPOB(L)(`bcast`(m))
1 **if** has-label(m)
2 **then** $l \leftarrow$ label(m)
3 ProtectedBlock(*need-token*$_l$, *door*$_l$, bcastop(m, LABELED))
4 **else** bcastop(m, UNLABELED)

The broadcast of a labeled message further requires that all processes in the ring deliver the message before the token is released. To ensure this, the broadcast waits for acknowledgments from all the processes in the ring before completing. A separate deliver thread is used to receive broadcasts and send acknowledgments. Unlabeled messages do not require acknowledgments.

bcastop(m,type)
{Broadcast}
1 **forall** $q \in P$
2 **do** $\text{SEND}(p, q, [\text{MESSAGE}, m])$
3 **if** *type* == LABELED
{Wait for acknowledgment}
4 **then forall** $q \in P$
5 **do** $(q, p, [\text{ACK}]) \dashv \text{recv}()$

The deliver operation must acknowledge labeled messages and not acknowledge unlabeled messages.

TokenPOB(L)(`deliver`())

```
1  (q, p, [MESSAGE, m]) ⊣ recv()
2  if has-label(m)
3      then SEND(p, q, [ACK])
4  return m
```

Reads and writes are mapped through with an identity transform. $\widehat{x}$ is used to emphasize that the platform has changed.

```
TokenPOB(L)(read(x))
1  return x̂
```

```
TokenPOB(L)(write(x, v))
1  x̂ ← v
```

The parameter L of the transform TokenPOB(L) is needed because a token is created for each label. To clarify the presentation, we will often drop the parameter and just write TokenPOB, as L will be provided by the specification POB-Cluster(L).

5.2 Correctness of TokenPOB implementation

Let (P, J) be a program compatible with the Partial-order-broadcast cluster platform.

Theorem 5.2.1. (TokenPOB(L)(P), TokenPOB(L)(J), ThreadedNetwork) *implements* (P, J, POB-Cluster(L))

Begin Proof

We use the general outline of memory consistency proofs from Section 3.3.1.

Assume: Let $\widehat{C}$ be a computation in (TokenPOB(P), TokenPOB(J), ThreadedNetwork) and let C be a computation such that $\widehat{C} \underset{\text{TokenPOB}^{*-1}}{\models} C$.

We must show that POB-Cluster(L)[C]. To do this we provide witnesses that satisfy the requirements of the definition.

Build: To satisfy POB-Cluster(L)[C] using Definition 5.0.4 we build a message order $(M, \underset{MO}{\longrightarrow})$ and a collection of view orders $\langle\ (O|p, \underset{L_p}{\longrightarrow}) : p \in P :\ \rangle$. To do so, we choose a

collection of witness orders $\langle\ (O|\widehat{p}, \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow}) : \widehat{p} \in P :\ \rangle$ to $\mathsf{ThreadedNetwork}[\widehat{C}]$. We also use the $\underset{\widehat{\text{HappensBefore}}_{\widehat{C}}}{\longrightarrow}$ order that is constructed directly from the computation and its components, described in Section 3.1.2.

We first build the message order, $\underset{MO}{\longrightarrow}$, which is constructed using the orders guaranteed by $\mathsf{ThreadedNetwork}[\widehat{C}]$. The message order is the $\underset{\widehat{\text{HappensBefore}}}{\longrightarrow}$ order on the broadcast operations. The tokens will make deliveries agree using the global $\underset{\widehat{\text{HappensBefore}}}{\longrightarrow}$ order. The $\underset{\widehat{\text{HappensBefore}}}{\longrightarrow}$ order will induce a stronger order on the delivers of labeled broadcasts, than it will on the delivers of unlabeled broadcasts, which will induce. Defining the orders enforced on the labeled broadcasts, $\underset{MORG}{\longrightarrow}$, and unlabeled broadcasts, $\underset{MORU}{\longrightarrow}$, separately will simplify our proofs. The message order will be the transitive closure of the union of both of these induced orders.

$$
\begin{aligned}
&\underset{MORG}{\longrightarrow} = \\
&\quad \{(m_1,m_2) : \\
&\qquad \text{has-label}(m_1) \wedge \text{has-label}(m_2) \wedge \Big(\text{label}(m_1) = \text{label}(m_2)\Big) \\
&\qquad \wedge \text{TokenPOB}(\texttt{bcast}(m_1)).\,\text{bcastop} \\
&\qquad\quad \underset{\widehat{\text{HappensBefore}}}{\longrightarrow} \text{TokenPOB}(\texttt{bcast}(m_2)).\,\text{bcastop}\} \\
&\underset{MORU}{\longrightarrow} = \\
&\quad \{(m_1,m_2) : \\
&\qquad \text{TokenPOB}(\texttt{bcast}(m_1)).\,\text{bcastop} \underset{\widehat{\text{prog}}}{\longrightarrow} \text{TokenPOB}(\texttt{bcast}(m_2)).\,\text{bcastop}\} \\
&\underset{MOR}{\longrightarrow} = \underset{MORG}{\longrightarrow} \cup \underset{MORU}{\longrightarrow} \\
&\underset{MO}{\longrightarrow} = (\underset{MOR}{\longrightarrow})^{+}
\end{aligned}
$$

The next orders to build are the local view orders. As noted earlier, for build steps we will abuse our notation. For a total order $(O, \underset{X}{\longrightarrow})$, we denote its induced sequence by X. For each $p \in P$, construct the sequence L_p as follows:

- Take the sequence $\widehat{L}_{\widehat{p}}$ formed from the witness total order $(\widehat{O}|\widehat{p}, \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow})$ to $\mathsf{ThreadedNetwork}[\widehat{C}]$ and remove:

1. all operations of the $need_l$, $door_l$ variables
2. all send and receive operations except: the first send of each bcastop() call, and the only recv of each deliver call.

- Associate the target level operations with specification level ones by defining $\underset{construct}{\sim}$ as follows:
  1. Each $\frac{\text{READ}(x)}{v}$ is in the transformation TokenPOB($\frac{\texttt{read}(x)}{v}$).
     Let $\frac{\text{READ}(x)}{v} \underset{construct}{\sim} \frac{\texttt{read}(x)}{v}$.
  2. Each $\text{WRITE}(x, v)$ is in the transformation TokenPOB($\texttt{write}(x, v)$).
     Let $\text{WRITE}(x, v) \underset{construct}{\sim} \texttt{write}(x, v)$.
  3. Each remaining $\text{SEND}(s, d, m) = \text{TokenPOB}(\texttt{bcast}(m))._1\text{SEND}()$. Recall that this notation denotes the first instance of an operation in that instance of the subroutine, in program order.
     Let $\text{SEND}(s, d, m) \underset{construct}{\sim} \texttt{bcast}(m)$.
  4. Each remaining $\frac{\text{RECV}()}{s, d, m} = \text{TokenPOB}(\frac{\texttt{deliver}()}{m}).\text{RECV}()$.
     Let $\frac{\text{RECV}()}{m} \underset{construct}{\sim} \frac{\texttt{deliver}()}{m}$.
  5. Replace each target level operation with its associated specification level operation

Verify: We must now show that the constructed orders, $\underset{MO}{\longrightarrow}$, $\langle\ \underset{L_p}{\longrightarrow} : p \in P :\ \rangle$ are witnesses to ThreadedNetwork$[C]$. This is done by verifying that the constructed orders satisfy each conjunct of Definition 5.0.4.

The construction yields a collection of total orders $(O|p, \underset{L_p}{\longrightarrow})$. These orders are valid by the fact that projecting on to subsets of objects preserves validity and that all sequences of bcast and deliver are valid. The constraints of the definition are satisfied by the following lemmas.

Constraint	Lemma
$(M, \underset{MO}{\longrightarrow})$ partial order	Lemma 5.2.2
$\dfrac{\texttt{deliver}()}{m} \in O \implies m \in M$	Lemma 5.2.3
a message is delivered only once per processor	Lemma 5.2.3
$\forall l \in L : (M\|l, \underset{MO}{\longrightarrow})$ is total	Lemma 5.2.6
$\texttt{bcast}(m_1) \underset{\text{prog}}{\longrightarrow} \texttt{bcast}(m_2) \implies m_1 \underset{MO}{\longrightarrow} m_2$	Lemma 5.2.5
$\texttt{bcast}(m), \dfrac{\texttt{deliver}()}{m} \in O\|p \implies \texttt{bcast}(m) \underset{L_p}{\longrightarrow} \dfrac{\texttt{deliver}()}{m}$	Lemma 5.2.4
$m_1 \underset{MO}{\longrightarrow} m_2 \wedge \dfrac{\texttt{deliver}()}{m_2} \in O\|p \implies \dfrac{\texttt{deliver}()}{m_1} \underset{L_p}{\longrightarrow} \dfrac{\texttt{deliver}()}{m_2}$	Lemma 5.2.9
$\mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O\|p]$	Lemma 5.2.10
$(O\|p, \underset{L_p}{\longrightarrow})$ is a valid total order	by construction

By above lemmas, the constructed $\underset{MO}{\longrightarrow}, \underset{L_p}{\longrightarrow} : p \in P$ are witness orders to $\mathsf{POB\text{-}Cluster}(L)[C]$ as required. By definition the transformation is an implementation.

End Proof

Let C and $\widehat{C}$ be computations such that

$\widehat{C} \in (\text{TokenPOB}(P), \text{TokenPOB}(J), \mathsf{ThreadedNetwork})$ and $\widehat{C} \underset{\text{TokenPOB}^{*-1}}{\models} C$. This matches the **Assume** step of Theorem 5.2.1.

Let $(M, \underset{MO}{\longrightarrow})$ and $\langle\, (O|p, \underset{L_p}{\longrightarrow}) : p \in P : \rangle$ be constructed from the witness orders $\langle\, (O|\widehat{p}, \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow}) : \widehat{p} \in P : \rangle$. This matches the **Build** step of Theorem 5.2.1.

Lemma 5.2.2. $(M, \underset{MO}{\longrightarrow})$ *is a partial order*

Begin Proof

Let $\underset{X}{\longrightarrow} = \{(m_1, m_2) :$

$$\text{TokenPOB}(K)(\texttt{bcast}(m_1)).\,\text{bcastop} \underset{\widehat{\text{HappensBefore}}}{\longrightarrow} \text{TokenPOB}(K)(\texttt{bcast}(m_2)).\,\text{bcastop}\}$$

Let B be the set of bcastop instances.

Observe that $(M, \underset{X}{\longrightarrow})$ is a partial order since $(B, \underset{\widehat{\text{HappensBefore}}}{\longrightarrow})$ is a partial order and mapping bcastops to their messages gives a bijection between B and M. $\underset{\widehat{\text{HappensBefore}}}{\longrightarrow}$ is unique for each computation C, and does not change meanings when considering different subsets of O_C.

$$
\begin{array}{l}
\quad \forall m_1, m_2 \in M : (m_1 \underset{MO}{\longrightarrow} m_2) \implies (m_1 \underset{MO}{\longrightarrow} m_2) \\
\Longrightarrow \left\{ \quad \text{construction of } \underset{MO}{\longrightarrow} \quad \right\} \\
\quad \forall m_1, m_2 \in M : (m_1 \underset{MO}{\longrightarrow} m_2) \\
\qquad \Longrightarrow (\text{TokenPOB}(K)(\texttt{bcast}(m_1)).\,\text{bcastop} \\
\qquad\qquad \underset{\widehat{\text{HappensBefore}}}{\longrightarrow} \text{TokenPOB}(K)(\texttt{bcast}(m_2)).\,\text{bcastop}) \\
\Longrightarrow \left\{ \quad \text{construction of } \underset{X}{\longrightarrow} \quad \right\} \\
\quad \forall m_1, m_2 \in M : (m_1 \underset{MO}{\longrightarrow} m_2) \implies (m_1 \underset{X}{\longrightarrow} m_2) \\
\Longrightarrow \{ \quad \text{definition of } \mathsf{Extends} \} \\
\quad \mathsf{Extends}[\underset{X}{\longrightarrow}, \underset{MO}{\longrightarrow}, M]
\end{array}
$$

Therefore since $(M, \underset{X}{\longrightarrow})$ is irreflexive, $(M, \underset{MO}{\longrightarrow})$ is irreflexive and since $\underset{MO}{\longrightarrow}$ is transitive by definition, $(M, \underset{MO}{\longrightarrow})$ is a partial order.

End Proof

The following lemma shows that each message is only delivered once per processor. This is an object restriction rather than a memory consistency constraint, but it must still be verified.

Lemma 5.2.3. $\frac{\texttt{deliver}()}{m} \in O \implies m \in M$ *and*
$\forall p \in P$: *there is at most one* $\frac{\texttt{deliver}()}{m} \in O|p$

Begin Proof

Let $\frac{\texttt{deliver}()}{m} \in O$, then

$$\text{TokenPOB}(\frac{\texttt{deliver}()}{m}).\text{RECV}() \xleftarrow[\widehat{\text{MessageOrder}}]{1.} \text{TokenPOB}(\texttt{bcast}(m)).\text{SEND}()$$

1. Relaxing notation, TokenPOB($\texttt{bcast}(m)$).SEND() is the send that matches the single receive in the deliver. Remember that $\xrightarrow[\widehat{\text{MessageOrder}}]{}$ is the order defined for the definition of ThreadedNetwork and is distinct from $\xrightarrow[MO]{}$.

therefore $\texttt{bcast}(m) \in O \implies m \in M$.

A message can only delivered as many times as it is sent to a processor, since it must be received each time. We can see by the code that each message is only sent to each processor once, so it can be delivered at most once per processor.

End Proof

The following lemma shows that locally broadcasted messages cannot be locally delivered before they are broadcasted. This type of property is necessary due to the weak notion of time that we use. Some weak memory models may allow some "impossible" situations to occur. Another possible violation of this property is an implementation that guesses the message that will be delivered, and rolls back the effects of the guess if it turns out to be wrong. This would be similar to branch prediction in a microprocessor.

Lemma 5.2.4. $\texttt{bcast}(m), \dfrac{\texttt{deliver}()}{m} \in \widehat{O}|\widehat{p} \implies \texttt{bcast}(m) \xrightarrow[L_p]{} \dfrac{\texttt{deliver}()}{m}$

Begin Proof

Let $\texttt{bcast}(m), \dfrac{\texttt{deliver}()}{m} \in O|p$. Then the transformation of $\dfrac{\texttt{deliver}()}{m}$ must contain a recv, $\text{TokenPOB}(\dfrac{\texttt{deliver}()}{m}).\text{RECV}() = \dfrac{\text{RECV}()}{s,d,m}$. This operation must have a corresponding send operation satisfying $\text{SEND}(s,d,m) \underset{\widehat{\text{MessageOrder}}}{\longrightarrow} \dfrac{\text{RECV}()}{s,d,m}$.

A send can only be produced by its corresponding bcast, so:

$$\text{SEND}(s,d,m) \in \text{TokenPOB}(\texttt{bcast}(m))$$

Then since $\mathsf{Extends}[\underset{\widehat{\text{HappensBefore}}}{\longrightarrow}, \underset{\widehat{\text{MessageOrder}}}{\longrightarrow}, \widehat{O}|\widehat{p}]$:

$\text{TokenPOB}(\texttt{bcast}(m)).\text{SEND}() \underset{\widehat{\text{MessageOrder}}}{\longrightarrow} \text{TokenPOB}(\dfrac{\texttt{deliver}()}{m}).\text{RECV}()$ since, we have $\texttt{bcast}(m), \dfrac{\texttt{deliver}()}{m} \in O|p$ this implies that the transformations are in $\widehat{O}|\widehat{p}$ and:

$$\text{TokenPOB}(\texttt{bcast}(m)).\text{SEND}(\widehat{p},\widehat{p},m) \xrightarrow[\widehat{L}_{\widehat{p}}]{} \text{TokenPOB}(\frac{\texttt{deliver}()}{m}).\frac{\text{RECV}()}{\widehat{p},\widehat{p},m}$$

then this also holds for the first send in the broadcast:

$$\text{TokenPOB}(\texttt{bcast}(m))._1\text{SEND}() \xrightarrow[\widehat{L}_{\widehat{p}}]{} \text{TokenPOB}(\frac{\texttt{deliver}()}{m}).\frac{\text{RECV}()}{\widehat{p},\widehat{p},m}$$

Since $\text{TokenPOB}(\texttt{bcast}(m))._1\text{SEND}() \underset{construct}{\sim} \texttt{bcast}(m)$

and $\text{TokenPOB}(\dfrac{\texttt{deliver}()}{m}).\dfrac{\text{RECV}()}{\widehat{p},\widehat{p},m} \underset{construct}{\sim} \dfrac{\texttt{deliver}()}{m}$ then by construction:

$$\texttt{bcast}(m) \xrightarrow[L_p]{} \frac{\texttt{deliver}()}{m}$$

End Proof

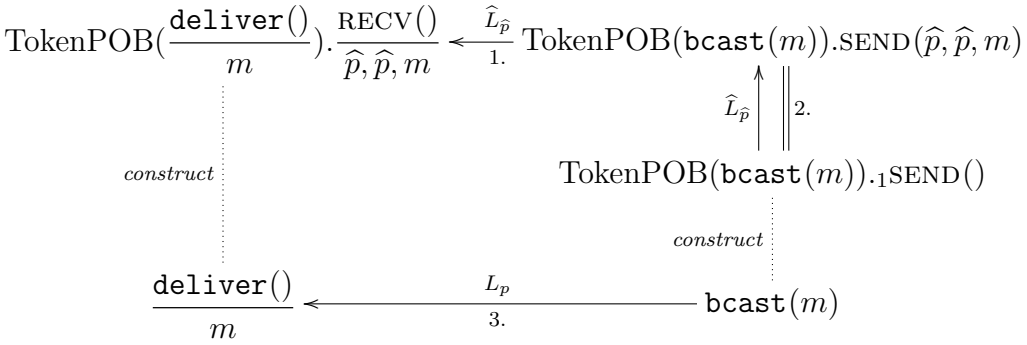
Since the above proof can be difficult to read, we use the diagram format described

in Section 3.3.2 to make the reasoning clearer and more readable.

Diagram proof of Lemma 5.2.4

Begin Proof

Let $\texttt{bcast}(m), \dfrac{\texttt{deliver}()}{m} \in O|p$ then this implies that:



1. The $\dfrac{\text{RECV}()}{\widehat{p},\widehat{p},m}$ must have a corresponding $\text{SEND}(p,p,m)$, this send could only have come from the transformation of $\texttt{bcast}(m)$. Since they are ordered by $\underbrace{\longrightarrow}_{\widehat{\text{MessageOrder}}}$ they must also be ordered by $\underset{\widehat{L}_{\widehat{p}}}{\longrightarrow}$.

2. Since $(\text{TokenPOB}(\texttt{bcast}(m)), \underset{\text{prog}}{\longrightarrow})$ is a total order, any send in it must either be the first send or follow it.

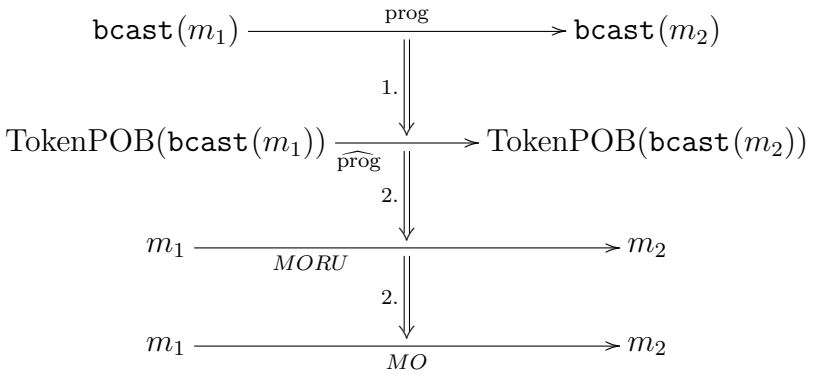
3. By construction, the associated operations will be ordered similarly in the constructed $\underset{L_p}{\longrightarrow}$ orders as in the $\underset{\widehat{L}_{\widehat{p}}}{\longrightarrow}$ ones.

End Proof

The next lemma shows that message order follows program order.

Lemma 5.2.5. $\texttt{bcast}(m_1) \underset{\text{prog}}{\longrightarrow} \texttt{bcast}(m_2) \implies m_1 \underset{MO}{\longrightarrow} m_2$

Begin Proof



1. The transformations preserve program order

2. By the definition of $\underset{MORU}{\longrightarrow}$ and by the definition of $\underset{MO}{\longrightarrow}$

End Proof

The following lemma shows that the total ordering of messages with the same label is enforced.

Lemma 5.2.6. $\forall\, l \in L : (M|l, \underset{MO}{\longrightarrow})$ *is a total order*

Begin Proof

Let $m_1, m_2 \in M|l$, then:

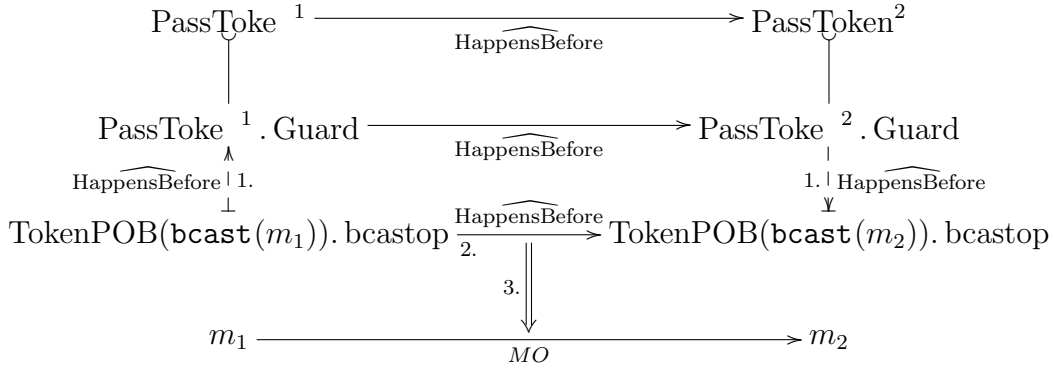
1. Property of the ProtectedBlock and Guard , $\text{PassToken}^1 \neq \text{PassToke}^{\ 2}$ for a similar reason.

It is clear that either

$$\text{PassToken}^1(l) \underset{\widehat{\text{HappensBefore}}}{\longrightarrow} \text{PassToken}^2(l) \text{ or } \text{PassToken}^2(l) \underset{\widehat{\text{HappensBefore}}}{\longrightarrow} \text{PassToken}^1(l)$$

since there is only one original send for each token, and each PassToken must be caused by the original send or the end of a previous PassToken.

Suppose WLOG that $\text{PassToken}^1 \underset{\widehat{\text{HappensBefore}}}{\longrightarrow} \text{PassToken}^2$ then:



1. By previous diagram

2. Must follow the order on the PassToken instances.

3. By the construction of $\underset{MORG}{\longrightarrow}$ and $\underset{MO}{\longrightarrow}$

Therefore $m_1 \underset{MO}{\longrightarrow} m_2$, and since $(M|l, \underset{MO}{\longrightarrow})$ is a partial order by Lemma 5.2.2, it must also be a total order.

End Proof

This lemma verifies that the delivers of labeled messages are ordered by their corresponding broadcasts.

Lemma 5.2.7.

$$\forall \texttt{bcast}(m) \in O, \frac{\texttt{deliver}()}{m} \in O :$$
$$\text{has-label}(m) \implies$$
$$\text{TokenPOB}(\frac{\texttt{deliver}()}{m}) \vdash \overset{\widehat{\text{HappensBefore}}}{\dashrightarrow} \text{TokenPOB}(\texttt{bcast}(m)).\,\text{bcastop}$$
$$\wedge\ \text{TokenPOB}(\frac{\texttt{deliver}()}{m}) \nvdash \overset{\widehat{\text{HappensBefore}}}{\dashrightarrow} \text{TokenPOB}(\texttt{bcast}(m)).\,\text{bcastop}$$

Begin Proof

We must show that the following diagram holds.

$$\begin{array}{ccc}
\text{TokenPOB}^1(\texttt{bcast}(m)).\text{SEND}(p,q,m) & \xrightarrow[\widehat{\text{MessageOrder}}]{1.} & \text{TokenPOB}^2(\frac{\texttt{deliver}()}{m}).\frac{\text{RECV}()}{p,q,m} \\
\Big\downarrow \widehat{\text{prog}} & & \Big\downarrow 1. \widehat{\text{prog}} \\
\text{TokenPOB}^1(\texttt{bcast}(m)).\frac{\text{RECV}()}{q,p,[\text{ACK}]} & \xleftarrow[2.]{\widehat{\text{MessageOrder}}} & \text{TokenPOB}^2(\frac{\texttt{deliver}()}{m}).\text{SEND}(\overset{q,p,}{}_{[\text{ACK}]})
\end{array}$$

1. The message sent in the transformation of the bcast must have been received by the transformation of some deliver. The transformation of the deliver must also contain a send of an acknowledgment.

2. Since the message send and recv must be related by message order, we must show that the acknowledgment received by TokenPOB($\texttt{bcast}(m)$) is actually from the deliver in step 1 and not a different one.

   Suppose that the [ACK] message came from the transformation of another deliver $\frac{\texttt{deliver}()}{m'}$, then it must have occurred at the same processor q and $\frac{\texttt{deliver}()}{m'} \xrightarrow[\text{prog}]{} \frac{\texttt{deliver}()}{m}$ since the [ACK] messages are received in FIFO order. This $\frac{\texttt{deliver}()}{m'}$ then must have also received its [ACK] from a previous $\frac{\texttt{deliver}()}{m''}$. Proceeding inductively, this would imply an infinite chain of previous deliver operations, contradiction.

End Proof

This lemma shows that delivers agree with $\underset{MOR}{\longrightarrow}$ order. The next lemma will apply the transitive closure to this property to obtain the proof for $\underset{MO}{\longrightarrow}$.

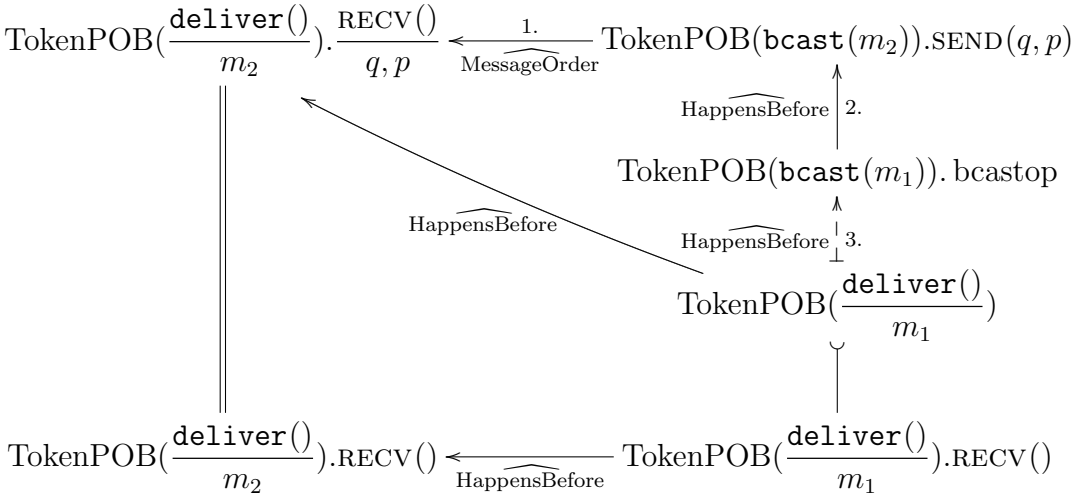
Lemma 5.2.8. *If* $m_1 \underset{MOR}{\longrightarrow} m_2$ *and* $\frac{\texttt{deliver}()}{m_2} \in O|p$ *then*

$$\frac{\texttt{deliver}()}{m_1} \in O|p \wedge \text{TokenPOB}(\frac{\texttt{deliver}()}{m_1}).\text{RECV}() \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow} \text{TokenPOB}(\frac{\texttt{deliver}()}{m_2}).\text{RECV}()$$

Begin Proof

Let $m_1, m_2 \in M$ such that $m_1 \underset{MOR}{\longrightarrow} m_2$. Then there are two cases for m_1 and m_2:

Case 1 : group ordered updates $m_1 \underset{MORG}{\longrightarrow} m_2$ then



1. The corresponding send of the recv must be from some $q \in P$.
2. Since $m_1 \underset{MORG}{\longrightarrow} m_2$ this holds by the construction of $\underset{MORG}{-}$
3. Lemma 5.2.7

Case 2 : program ordered updates $m_1 \underset{MOGU}{\longrightarrow} m_2$ then

$$\begin{array}{ccc} \text{TokenPOB}(\texttt{bcast}(m_1)).\text{SEND}() & \xrightarrow{\widehat{\text{prog}}} & \text{TokenPOB}(\texttt{bcast}(m_2)).\text{SEND}() \\ & \overset{1.}{\Downarrow} & \\ \text{TokenPOB}(\dfrac{\texttt{deliver}()}{m_1}).\text{RECV}() & \xrightarrow[\widehat{\text{HappensBefore}}]{} & \text{TokenPOB}(\dfrac{\texttt{deliver}()}{m_2}).\text{RECV}() \end{array}$$

1. By the FIFO property of ThreadedNetwork.

The two cases imply:

$$\begin{aligned} & m_1 \underset{MO}{\longrightarrow} m_2 \wedge \frac{\texttt{deliver}()}{m_2} \in O|p \\ \Longrightarrow\ & m_1 (\underset{MOR}{\longrightarrow})^+ m_2 \\ \Longrightarrow\ & \text{TokenPOB}(\frac{\texttt{deliver}()}{m_1}).\text{RECV}() (\underset{\widehat{\text{HappensBefore}}}{\longrightarrow})^+ \text{TokenPOB}(\frac{\texttt{deliver}()}{m_2}).\text{RECV}() \\ \equiv\ & \text{TokenPOB}(\frac{\texttt{deliver}()}{m_1}).\text{RECV}() \underset{\widehat{\text{HappensBefore}}}{\longrightarrow} \text{TokenPOB}(\frac{\texttt{deliver}()}{m_2}).\text{RECV}() \\ \Longrightarrow\ & \{ \quad \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow} \text{ extends } \underset{\widehat{\text{HappensBefore}}}{\longrightarrow} \text{ on } \widehat{O}|\widehat{p} \quad \} \\ & \text{TokenPOB}(\frac{\texttt{deliver}()}{m_1}).\text{RECV}() \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow} \text{TokenPOB}(\frac{\texttt{deliver}()}{m_2}).\text{RECV}() \end{aligned}$$

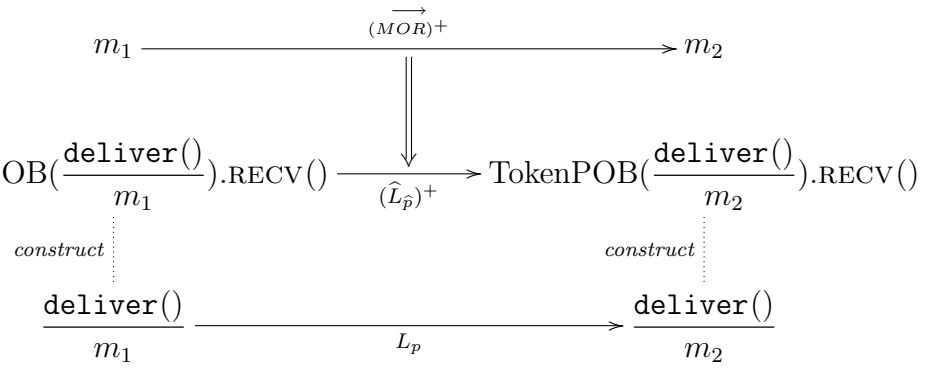
as required.

End Proof

Lemma 5.2.9. $m_1 \xrightarrow[MO]{} m_2 \wedge \dfrac{\texttt{deliver}()}{m_2} \implies \dfrac{\texttt{deliver}()}{m_1} \xrightarrow[L_p]{} \dfrac{\texttt{deliver}()}{m_2}$

Begin Proof

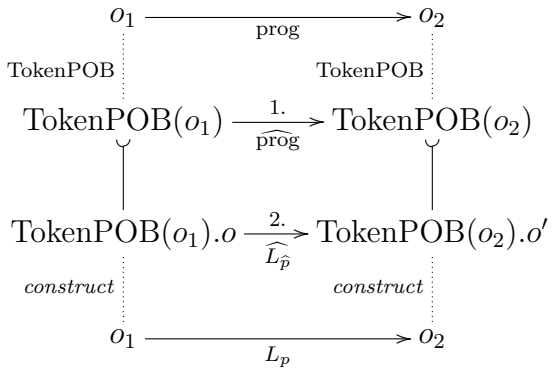
Let $m_1 \xrightarrow[MO]{} m_2$, then we have by Lemma 5.2.8 :



End Proof

Lemma 5.2.10. Extends$[\xrightarrow[L_p]{}, \xrightarrow[\text{prog}]{}, O|p]$

Begin Proof



1. Can be observed from the definition of the transformation.

2. Where o and o' are the target level operations in the set that remain after the Short() step of the construction. Holds since $A \to B \equiv (\forall a \in A, b \in B : a \to b)$

End Proof

5.3 QueuePOB implementation

The queue implementation takes a different approach to enforcing agreement. Instead of strongly ordering the deliveries in time, it enforces an agreement on the order that deliveries are performed. This implementation is based on Attiya and Welch's timestamp total order broadcast implementation presented in their textbook [8] .

Priority queues store received messages in timestamp order. There is one queue per message label. Priority queues will ensure that when a message is dequeued, it has the least timestamp out of the remaining messages with that label. If we further ensure that all messages with a smaller or equal timestamp have been received and enqueued before dequeuing a message, then all processors will deliver messages with the same label in the same total order. Then to satisfy FIFO delivery order, we impose the additional requirement that every message delivered from a processor must be the message from that processor with the smallest undelivered timestamp. Because messages are spread across the multiple priority queues, FIFO delivery order is not automatically enforced as it was in the token implementation.

This implementation has the feature that it requires no extra threads. Each thread is transformed to obtain a new thread for the ThreadedNetwork platform. The first piece is the transformation of `bcast`(*message*). Since the actual broadcast requires modifying the queue, we pass this responsibility on to the deliver function to avoid race conditions and more complicated synchronization.

QueuePOB(L)(`bcast`(*message*))
1 SEND($p, p,$ [LOCAL-BROADCAST-REQUEST, *message*])

The deliver function is the workhorse of the transformation. It handles the actual broadcasting details as well as all of the background maintenance work.

QueuePOB(L)(`deliver`())

```
    {   g can be ⊥   }
1   while ¬∃g : CanDeliver(queue_g)
2         do HandleMessage()
3   choose g : CanDeliver(queue_g)
4      then m ← EXTRACTMIN(queue_g)
5           counter[m.src] ← m.counter
6           return m
```

The HandleMessage function handles all the background maintenance work, sending timestamp updates, modifying the timestamp and counter arrays, and queueing messages.

```
HandleMessage_p()
 1  message ← RECV()
 2  case message of:
 3  [LOCAL-BROADCAST-REQUEST, message]
 4     then T[p] ← T[p] + 1
 5          local-counter ← local-counter +1
 6          prepmessage ← [FLEX-ORDERED-MESSAGE, message, T[p], local-counter, p]
 7          ProcessMessage(prepmessage)
 8          FifoBroadcast(prepmessage)
 9  [TS-UPDATE, timestamp, q]
10     then T[q] ← timestamp
11  [FLEX-ORDERED-MESSAGE, m, timestamp, counter, q]
12     then T[q] ← timestamp
13          ProcessMessage(message)
14          if timestamp > T[p]
15             then T[p] ← timestamp
16                  FifoBroadcast([TS-UPDATE,T[p],p])
```

Some extra logic is needed to handle message labels by adding them to the correct queue. Unlabeled messages are queued in the special queue $queue_{\perp}$. The notation $m.ts$ denotes the *timestamp* field of a FLEX-ORDERED-MESSAGE or TS-UPDATE messages m. The notation $m.src$ denotes the q, or message source field of a FLEX-ORDERED-MESSAGE. The notation $m.counter$ denotes the *counter* the program counter field of a FLEX-ORDERED-MESSAGE.

ProcessMessage(*message*)

```
1  if has-label(message)
2     then group = label(message)
3          ENQUEUE(queue_group, message)
4     else ENQUEUE(queue_⊥, message)
```

Then we come to the actual logic that decides when a message can be delivered.

```
CanDeliver(queue)
1  if ISEMPTY(queue)
2     then return FALSE
3     else m ← PEEKMIN(queue)
4          return (m.counter = counter[m.src] + 1) ∧ (∀q ∈ P : m.ts < T[q])
```

The implementation of FifoBroadcast relies on the underlying platform to provide FIFO properties.

```
FifoBroadcast_p(message)
1  forall q ∈ P \ {p}
2       do SEND(p, q, message)
```

Reads and writes are mapped through with an identity transform. $\widehat{x}$ is used to emphasize that the platform has changed.

```
QueuePOB(L)(read(x))
1  return x̂
```

```
QueuePOB(L)(write(x, v))
1  x̂ ← v
```

The parameter L of the transform QueuePOB(L) is needed as a priority queue is created for each label. To clarify the presentation, we will often drop the parameter and just write QueuePOB, as L will be provided by the specification POB-Cluster(L).

5.4 Correctness of QueuePOB implementation

Let (P, J) be a program compatible with the Partial-order-broadcast cluster platform.

Theorem 5.4.1. $(\text{QueuePOB}(L)(P), \text{QueuePOB}(L)(J), \mathsf{ThreadedNetwork})$ *implements* $(P, J, \mathsf{POB\text{-}Cluster}(L))$

Begin Proof

We use the general outline of memory consistency proofs from Section 3.3.1.

Assume: Let $\widehat{C}$ be a computation in $\mathcal{C}\left(\text{QueuePOB}(P), \text{QueuePOB}(J), \mathsf{ThreadedNetwork}\right)$ and let C be a computation such that $\widehat{C} \underset{\text{QueuePOB}^{*-1}}{\models} C$.

We must show $\mathsf{POB\text{-}Cluster}(L)[C]$. To do this we construct witness orders that satisfy the requirements of Definition 5.0.4.

Build: We define a timestamp order $\underset{ts}{\longrightarrow}$ on messages. Lexicographic order is used to break ties between messages coming from different processes. The transformation ensures that for all m, $(m.ts, m.src)$ is unique.

$$m_1 \underset{ts}{\longrightarrow} m_2 \stackrel{\text{def}}{=} (m_1.ts, m_1.src) < (m_2.ts, m_2.src)$$

Timestamp order is a total order on messages, however, we allow our implementation to deliver some messages out of timestamp order, weakening it. We construct the message order of $\underset{MO}{\longrightarrow}$ by weakening the timestamp total order to allow the same order of deliveries as the implementation.

$$m_1 \underset{MOR}{\longrightarrow} m_2 \stackrel{\text{def}}{=} \Big(\big(\text{label}(m_1) = \text{label}(m_2)\big) \vee \big(m_1.src = m_2.src\big)\Big) \wedge m_1 \underset{ts}{\longrightarrow} m_2 \tag{5.4.1}$$

$$\underset{MO}{\longrightarrow} \stackrel{\text{def}}{=} (\underset{MOR}{\longrightarrow})^{+} \tag{5.4.2}$$

For build steps we will abuse our notation. For a total order $(O, \underset{X}{\longrightarrow})$, we denote its induced sequence by X. For each p, construct the sequence L_p as follows:

- Take the sequence $\widehat{L_{\widehat{p}}}$ formed from the witness total order $(\widehat{O}|\widehat{p}, \underset{\widehat{L_{\widehat{p}}}}{\longrightarrow})$ to ThreadedNetwork$[\widehat{C}]$ and form $Short(\widehat{L}_{\widehat{p}})$ by removing:

  1. all the operations of on the T, *counter*, *local-counter* variables, the $queue_l$ priority queues for each label and the $queue_\perp$ priority queue for the unlabeled messages.
  2. all send and receive operations except: the send of a [LOCAL-BROADCAST-REQUEST].
  3. all queue operations except EXTRACTMIN() operations.

- Associate the target level operations with specification level ones by defining $\underset{construct}{\sim}$ as follows:

  1. Each $\frac{\text{READ}(x)}{v}$ is in the transformation QueuePOB($\frac{\texttt{read}(x)}{v}$).
     Let $\frac{\text{READ}(x)}{v} \underset{construct}{\sim} \frac{\texttt{read}(x)}{v}$.
  2. Each WRITE(x, v) is in the transformation QueuePOB($\texttt{write}(x, v)$).
     Let WRITE$(x, v) \underset{construct}{\sim} \texttt{write}(x, v)$.
  3. Each remaining SEND(s, d, m) is in the transformation QueuePOB($\texttt{bcast}(m)$).
     Let SEND$(s, d, m) \underset{construct}{\sim} \texttt{bcast}(s, d, m)$.
  4. Each $\frac{\text{EXTRACTMIN}(queue)}{m}$ operation is in the transformation QueuePOB($\frac{\texttt{deliver}()}{m}$).
     Let $\frac{\text{EXTRACTMIN}(queue)}{m} \underset{construct}{\sim} \frac{\texttt{deliver}()}{m}$.

- Let the sequence L_p be $Short(\widehat{L}_{\widehat{p}})$ with all the target level operations replaced by their associated specification level operations .

Verify: As for the token implementation, we must show that the constructed orders are witness orders to POB-Cluster$(L)[C]$.

The construction yields witness total orders $(O|p, \underset{L_p}{\longrightarrow})$ that are valid by the fact that projecting on to subsets of objects preserves validity and that all sequences of bcast and deliver are valid. The sequence $Short(\widehat{L}_{\widehat{p}})$ is valid for all objects except queues, so by replacing the EXTRACTMIN() operations with `deliver`() operations, we regain validity. The constraints of Definition 5.0.4 are satisfied by the following lemmas:

Constraint	Lemma
$(M, \underset{MO}{\longrightarrow})$ partial order	by construction
$\frac{\texttt{deliver}()}{m} \in O \implies m \in M$	Lemma 5.4.3
a message is delivered only once per processor	Lemma 5.4.2
$\forall l \in L : (M\|l, \underset{MO}{\longrightarrow})$ is total	Lemma 5.4.5
$\texttt{bcast}(m_1) \underset{\text{prog}}{\longrightarrow} \texttt{bcast}(m_2) \implies m_1 \underset{MO}{\longrightarrow} m_2$	Lemma 5.4.4
$\texttt{bcast}(m), \frac{\texttt{deliver}()}{m} \in O\|p \implies \texttt{bcast}(m) \underset{L_p}{\longrightarrow} \frac{\texttt{deliver}()}{m}$	Lemma 5.4.3
$m_1 \underset{MO}{\longrightarrow} m_2 \wedge \frac{\texttt{deliver}()}{m_2} \in O\|p \implies \frac{\texttt{deliver}()}{m_1} \underset{L_p}{\longrightarrow} \frac{\texttt{deliver}()}{m_2}$	Lemma 5.4.6
$\mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O\|p]$	Lemma 5.4.8
$(O\|p, \underset{L_p}{\longrightarrow})$ is a valid total order	by construction

By the above lemmas, the constructed orders are witnesses to $\mathsf{POB\text{-}Cluster}(L)[C]$ as required.

End Proof

Let C and $\widehat{C}$ be computations such that

$\widehat{C} \in (\text{QueuePOB}(P), \text{QueuePOB}(J), \mathsf{ThreadedNetwork})$ and $\widehat{C} \underset{\text{QueuePOB}^{*-1}}{\models} C$. This matches the **Assume** step of Theorem 5.2.1.

Let $(M, \underset{MO}{\longrightarrow})$ and $\langle\, (O|p, \underset{L_p}{\longrightarrow}) : p \in P :\, \rangle$ be constructed from the witness orders $\langle\, (O|\widehat{p}, \underset{\widehat{L}_{\widehat{p}}}{\longrightarrow}) : \widehat{p} \in P :\, \rangle$. This matches the **Build** step of Theorem 5.2.1.

Lemma 5.4.2. *Each message is delivered at most once per processor,*

Begin Proof

Each message is sent exactly once to each processor, where it is enqueued exactly once, therefore it can be dequeued at most once, therefore delivered at most once.

End Proof

The following lemma shows that locally sent messages cannot be delivered before they are sent.

Lemma 5.4.3. $\frac{\texttt{deliver}()}{m} \in O \implies m \in M$
and $\texttt{bcast}(m), \frac{\texttt{deliver}()}{m} \in \widehat{O}|\widehat{p} \implies \texttt{bcast}(m) \xrightarrow[L_p]{} \frac{\texttt{deliver}()}{m}$

Begin Proof

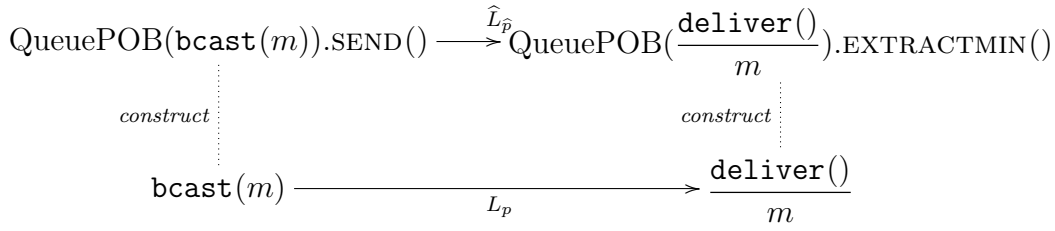
Let $\frac{\texttt{deliver}()}{m} \in O$.

We then follow the chain of operations that caused this operation to complete. The deliver requires a dequeue, which requires a corresponding enqueue. This enqueue must have occurred in a HandleMessage^1. Examining the code shows that HandleMessage^1 must have received the message from a different HandleMessage^2 instance. This leads us to an original bcast.

$$\begin{array}{ccccc}
\text{QueuePOB}(\frac{\texttt{deliver}()}{m}).\textsc{extractmin}() & \xleftarrow[\widehat{L_{\widehat{p}}}]{} & \text{HandleMessage}^1().\textsc{enqueue}([\overset{\text{ORD-MSG,}}{m}]) & & \\
 & & \uparrow \text{prog} & & \\
\text{HandleMessage}^2().\textsc{send}([\text{ORD-MSG}, m]) & \xrightarrow{\widehat{\text{MessageOrder}}} & \text{HandleMessage}^1().\frac{\textsc{recv}()}{[\text{ORD-MSG}, m]} & & \\
\uparrow \text{prog} & & & & \\
\text{HandleMessage}^2().\frac{\textsc{recv}()}{[\text{L-REQUEST}, m]} & \xleftarrow[\widehat{\text{MessageOrder}}]{} & \text{QueuePOB}(\texttt{bcast}(m)).\textsc{send}([\overset{\text{L-REQUEST,}}{m}]) & &
\end{array}$$

Therefore $\texttt{bcast}(m) \in O \wedge m \in M$.

Further suppose that $\texttt{bcast}(m), \dfrac{\texttt{deliver}()}{m} \in O|p$, then we can see from the above order that



Therefore $\texttt{bcast}(m) \xrightarrow[L_p]{} \dfrac{\texttt{deliver}()}{m}$.

End Proof

The following lemma shows that message order follows program order.

Lemma 5.4.4. $\texttt{bcast}(m_1) \xrightarrow[\text{prog}]{} \texttt{bcast}(m_2) \implies m_1 \xrightarrow[MO]{} m_2$

Begin Proof

We can observe from the code that messages broadcasted in program order must have strictly increasing timestamps, so:

$$\texttt{bcast}(m_1) \xrightarrow[\text{prog}]{} \texttt{bcast}(m_2) \implies (m_1.src = m_2.src \wedge m_1 \xrightarrow[ts]{} m_2) \implies m_1 \xrightarrow[MO]{} m_2$$

End Proof

Lemma 5.4.5. $\forall l \in L : (M|l, \underset{MO}{\longrightarrow})$ *is a total order*

Begin Proof

Let $m_1, m_2 \in M|\, l$ so that $m_1 \neq m_2$ then

$$\begin{aligned} & \text{label}(m_1) = \text{label}(m_2) \wedge \Big((m_1 \underset{ts}{\longrightarrow} m_2) \vee (m_2 \underset{ts}{\longrightarrow} m_1) \Big) \\ \Longrightarrow & \left\{ \quad \text{definition of } \underset{MO}{\longrightarrow} \quad \right\} \\ & (m_1 \underset{MO}{\longrightarrow} m_2) \vee (m_2 \underset{MO}{\longrightarrow} m_1) \end{aligned}$$

as required.

End Proof

Lemma 5.4.6. $m_1 \underset{MO}{\longrightarrow} m_2 \wedge \dfrac{\texttt{deliver}()}{m_2} \in O|p \implies \dfrac{\texttt{deliver}()}{m_1} \underset{L_p}{\longrightarrow} \dfrac{\texttt{deliver}()}{m_2}$

Begin Proof

From Lemma 5.4.7, we know that:

$$(m \underset{MOR}{\longrightarrow} m') \wedge \frac{\texttt{deliver}()}{m'} \in O|p \implies \frac{\texttt{deliver}()}{m} \underset{L_p}{\longrightarrow} \frac{\texttt{deliver}()}{m'}$$

Then:

$$\begin{aligned} & m_1 \underset{MO}{\longrightarrow} m_2 \\ \Longrightarrow\ & m_1 (\underset{MOR}{\longrightarrow})^+ m_2 \\ \Longrightarrow\ & \frac{\texttt{deliver}()}{m_1} (\underset{L_p}{\longrightarrow})^+ \frac{\texttt{deliver}()}{m_2} \\ \equiv\ & \frac{\texttt{deliver}()}{m_1} \underset{L_p}{\longrightarrow} \frac{\texttt{deliver}()}{m_2} \end{aligned}$$

End Proof

Lemma 5.4.7. $m_1 \underset{MOR}{\longrightarrow} m_2 \wedge \dfrac{\texttt{deliver}()}{m_2} \in O|p \implies \dfrac{\texttt{deliver}()}{m_1} \underset{L_p}{\longrightarrow} \dfrac{\texttt{deliver}()}{m_2}$

Begin Proof

Let $m_1 \underset{MOR}{\longrightarrow} m_2$ so that $\dfrac{\texttt{deliver}()}{m_2} \in O|p$ and let $\texttt{bcast}(m_1) \in O|q$ for some $q \in P$. Then QueuePOB($\dfrac{\texttt{deliver}()}{m_2}$).EXTRACTMIN() $\in \widehat{O}|\widehat{p}$.

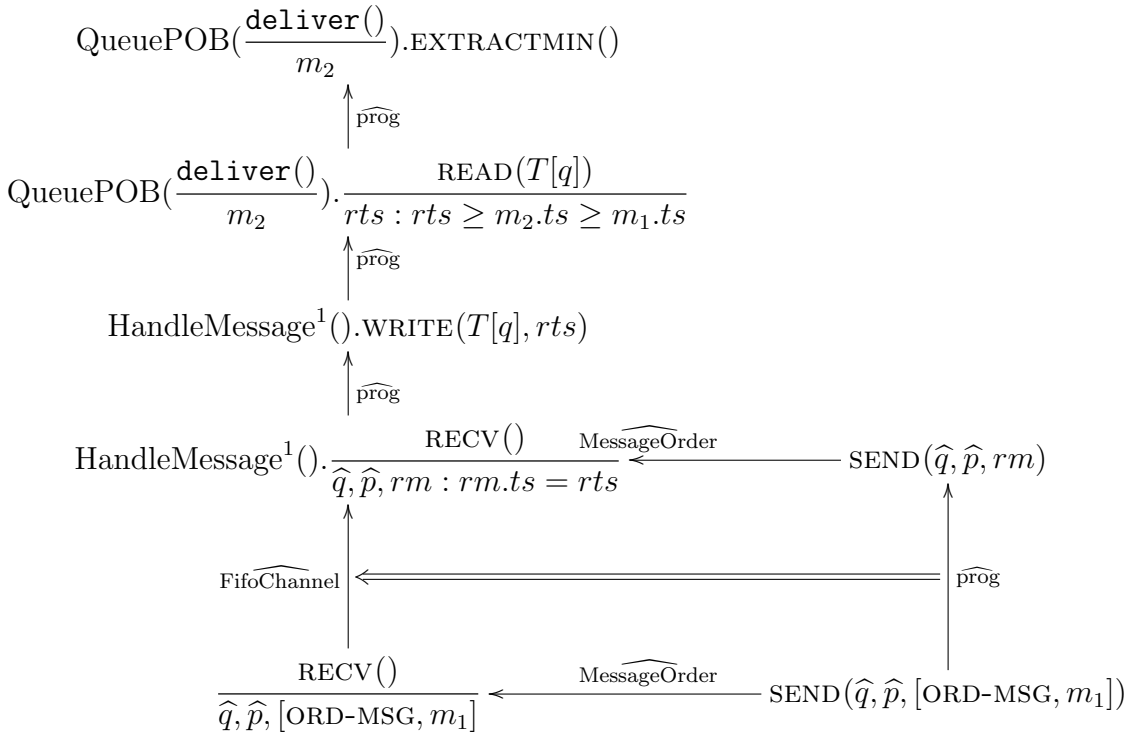
We then have

$$m_1 \underset{MOR}{\longrightarrow} m_2$$

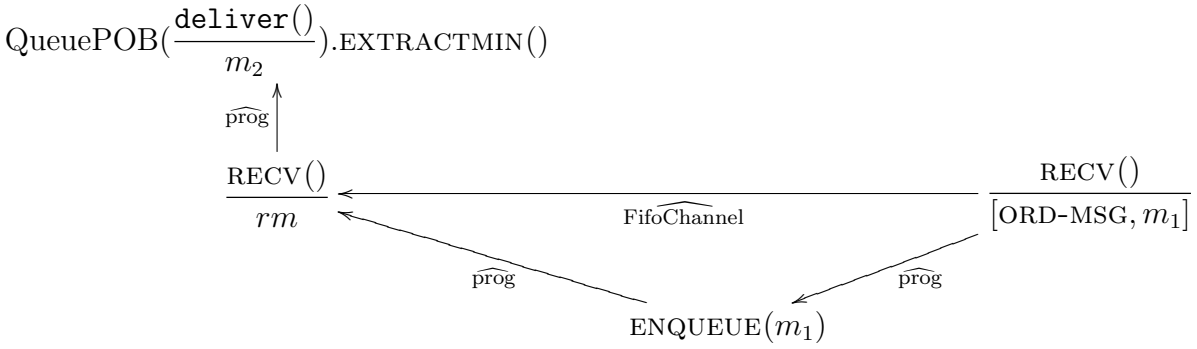
$$\implies m_1 \underset{ts}{\longrightarrow} m_2$$

$$\implies m_1.ts \leq m_2.ts$$

Before m_2 was dequeued, we must have read a greater or equal timestamp from $T[q]$. This must have been caused by some message rm with this timestamp from $T[q]$. This message rm is either m_1 or must have been sent after m_1. We consider the case that $m_1 \neq rm$.



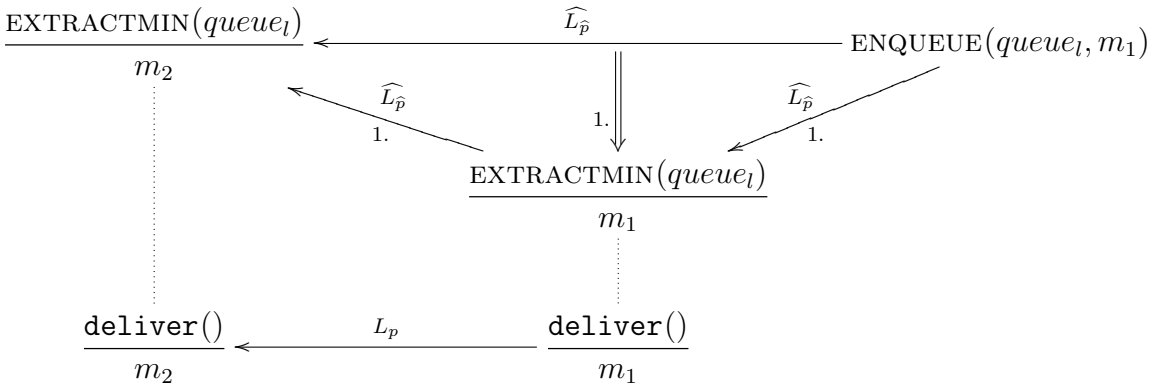
By examining the code, we can see that the recv of m_1 must have been followed by an enqueue, and since the recvs are executed by a single thread, this must have happened before $\frac{\text{RECV}()}{rm}$. Isolating an area of the previous diagram and adding in the enqueue gives:



Therefore $\text{ENQUEUE}(m_1) \xrightarrow[\text{prog}]{} \text{QueuePOB}(\frac{\texttt{deliver}()}{m_2}).\frac{\text{EXTRACTMIN}(queue_i)}{m_2}$. This result is the essential safety property of Attiya and Welch's timestamp total order broadcast algorithm. A process can only dequeue a message when it is sure that it has "seen" and enqueued all messages that might have a smaller timestamp.

Since $m_1 \underset{MOR}{-} m_2$ the definition gives two cases for m_1, m_2:

Case 1 same group: Then has-label(m_1), has-label(m_2) and label(m_1) = label(m_2). Let l = label(m_1) = label(m_2) and we have:



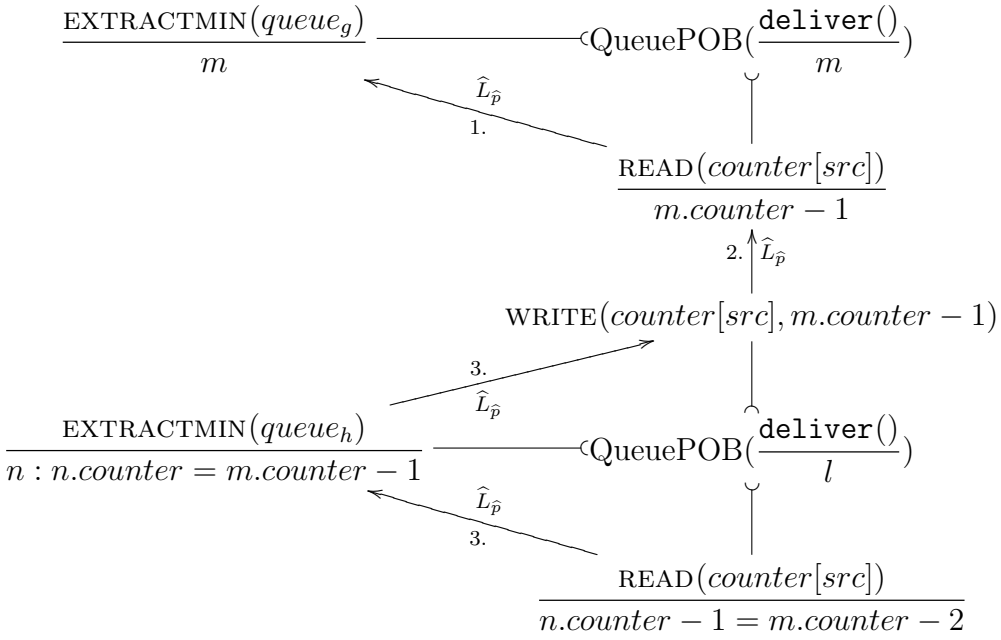
1. Since $m_1 \xrightarrow[ts]{} m_2$ the dequeue of m_1 is forced to occur before the dequeue of m_2, because of the min priority queue property. The dequeue of m_1 must come after the enqueue of m_1, because it is caused by that enqueue.

As required.

Case 2 same source: Then $m_1.src = m_2.src$. Let $src = m_1.src = m_2.src$.

$m_1.counter < m_2.counter$ since counter always increases and m_1 was bcast before m_2. Since $m_1.counter < m_2.counter$, the counter of m_1 must be in the set $[1, m_2.counter - 1]$. Also observe that $m_1.counter > 0$ so $m_1.counter \geq 1$ and $m_2.counter \geq 2$.

To deliver m_2 we must have $\frac{\texttt{read}(counter[src])}{m_2.counter - 1}$. We can see from the code that $counter[src]$ starts at 0 and can only be incremented by 1 each time a message from that source is delivered. This means that $counter[src]$ consecutively gets the value of each element in $[1, m_2.counter - 1]$ at some point before $\frac{\texttt{read}(counter[src])}{m_2.counter - 1}$. So, we must have delivered m_1 before m_2, as this diagram illustrates:

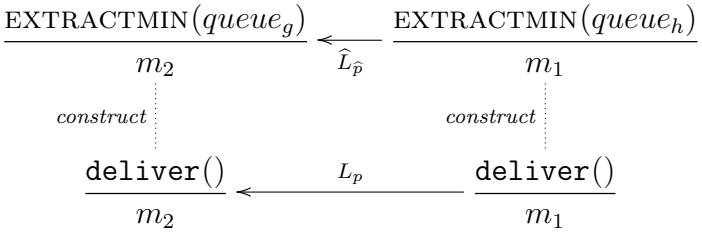


1. Examine the definition of QueuePOB($\frac{\texttt{deliver}()}{m}$). The CanDeliver call before a dequeue must return true.

2. If the read value is greater than the initial value of 0, the value must have been written by a previous deliver call.

3. Examine the definition of QueuePOB($\frac{\texttt{deliver}()}{n}$) the only write to counter occurs right after a dequeue. This leads to a situation similar to 1. except that the value read could be 0.

We can apply this result repeatedly to get:



As required.

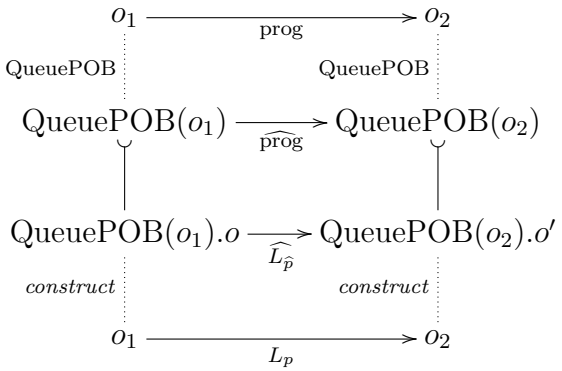
End Proof

The next lemma verifies the agreement condition.

Lemma 5.4.8. Extends$[\xrightarrow[L_p]{}, \xrightarrow[\text{prog}]{}, O|p]$

Begin Proof

The construction again provides a similar situation to TokenPOB:



End Proof

Chapter 6

Progress issues

Progress is a difficult issue to deal with in memory consistency frameworks. To avoid some of these difficulties, we prove very weak progress properties of our implementations in this chapter.

First define a *partial computation* to be a computation where at most one operation of each process is incomplete, and if that operation exists it is the last operation of that process. A partial computation is an *incomplete prefix* of a regular computation if there is a way to complete each incomplete operation to obtain a prefix of the regular computation. The trace transformation τ^* is extended to incomplete operations by taking all prefixes of the transformation of the operation execution, and this is further extended to partial computation.

We say that a partial computation has a completion if it can be extended to a computation where all the incomplete operations are complete. If the completion requires adding specification level operations, then these added operations should agree with the specification level program.

Completions eliminate the possibility of deadlock, including spin-loop deadlocks, which continue to produce operations but cannot make progress toward completion. They also justify our assumption of completed operations in the safety proofs. Some implementations will need stronger assumptions about the program to guarantee that completions always exist.

6.1 Slow Write, Fast Read and Fast Write, Slow Read transformation progress

Suppose we have a partial computation with some incomplete operations. The only way an operation can be blocked in this implementation is in the WaitWritesComplete call. If we can complete each incompleted WaitWritesComplete call, then we can complete each operation.

If operations can deadlock each other, it is important to consider a set of incomplete operations. In this case, we can complete each operation individually using the helper threads of each operation, so we can consider each operation individually.

This completion will not require any additional specification level operations to be added, this property is stronger than that of the later implementations.

Lemma 6.1.1. *WaitWritesComplete always has a completion.*

Proof sketch: WaitWritesComplete requires its delivery thread to update the counter to equal its current value. We complete this by finding the set of all undelivered local messages, this must be non-empty if the while loop is blocking progress. Out of these we take the one that was most recently broadcast, then find a chain of messages, including possibly non-local messages, that must be delivered before it.

We then run the delivery thread, delivering messages in the order of this chain. By the end of this, the count of delivered locally broadcasted messages equals the count of broadcasted local messages, since we have delivered all of them, and we can then complete the subroutine locally. □

6.2 Token partially ordered broadcast progress

Completions for the token implementation require that `deliver`() is called by a different thread than the one that calls `bcast`(), and that this other thread calls

`deliver`() infinitely often. Delivers cannot complete if all broadcasts have already been locally delivered, so there are no completions if a processor calls deliver, but no messages are available to be delivered.

These issues mean the progress property we prove is much weaker than the one shown in the previous section.

Lemma 6.2.1. *The transformation of* `bcast`() *always has a completion if for every other processor, its deliver thread has either partially completed a deliver call, or can reach another call by adding local operations.*

Proof sketch: Let the `bcast`() be from some process $p \in P$.

For an unlabeled message, `bcast`() can be completed by adding local operations without blocking.

For a labeled message, first we handle the case where the `bcast`() has already received the token. We can then locally complete the `bcast`() until it reaches the point where it waits for acknowledgements. We must then add completions to the delivery threads for these acknowledgements. For each delivery thread:

1. If the delivery thread is in the middle of a `deliver`() call, after the receive of the message, complete it. If this completion sent the acknowledgement required by the `bcast`(), stop.
2. Add local operations to the delivery thread until the next $\overline{\texttt{deliver}()}$ call is reached.
3. Begin the $\overline{\texttt{deliver}()}$ call by receiving a message from p, the process that called the `bcast`() we are completing. Return to step 1, this loop must handle all previously `bcast`()ed messages in FIFO order before being able to complete the `bcast`().

If the `bcast`() has not yet received the token, then we can find the process that is about to receive or already has the token. If it already has the token, complete this `bcast`() using the above procedure until this process passes on the token. Then go around the token ring, passing the token until it reaches the required bcast. Perform the appropriate handshake to start the bcastop phase of the `bcast`(), then complete as described above. □

Lemma 6.2.2. *The transformation of* `deliver`() *has a completion if there is a message available that can be delivered, or if it has already received a message.*

Proof sketch: If the `deliver`() has not received a message yet, add the receive of an available message. At this point, the `deliver`() can be locally completed. □

6.3 Queue partially ordered broadcast progress

As in the token implementation, completions will require that `deliver`() is called infinitely often. Also, as in the token implementation, we are unable to complete `deliver`() calls when there are no messages available to deliver.

Lemma 6.3.1. *The transformation of* `bcast`() *always has a completion.*

Proof. The `bcast`() in the queue implementation only performs a send, which does not block, so it completes. □

Lemma 6.3.2. *The transformation of* `deliver`() *always has a completion if there is a message available that can be delivered.*

Proof sketch: If the `deliver`() call has called CanDeliver, and it has returned true, then we can locally complete the operation. Otherwise, we will complete the deliver to return message m with the lowest timestamp that has not been delivered yet. Let q be the process that broadcasted m.

Either this message is in a queue, or it has not yet been received. If the message is not in a queue, run CanDeliver, if a receive is called, receive the next message from q in FIFO order. Repeat until m is in a queue. Now m must be at the top of some queue. We must satisfy the remaining CanDeliver conditions. To satisfy the timestamp property, obtain the set of processes r so that $T[r] < m.ts$.

For each of these processes r, there are three cases. If process r has sent a message with timestamp $T[r] \geq m.ts$ then run the loop until we receive the message. If process r is about to send this message, complete it to send the message, then loop until we receive it. Otherwise, process r has not received m yet, complete it to receive m, then loop until we receive it.

At this point we have received from each process a message n with $n.ts \geq m.ts$ so for all s in P, $T[p] \geq m.ts$.

Finally we must satisfy the program counter condition. All previous messages in program order must have a lower timestamp. m is the message not delivered with the lowest timestamp, so it must be the next one in the program sequence.

We can then complete the operation by delivering this lowest timestamped message. □

6.4 Issues with developing a fuller theory of progress

The notion of progress we prove is quite weak, but still does not provide as nice a composability as our safety proofs. In our token and queue implementations, the progress property depends on the program being able to call more deliver calls.

For the partial order broadcast implementations the progress was dependent on the way operations were called, due to our blocking operations. If they were changed to non-blocking operations by adding a "no message ready" return value, a stronger property could possibly be proved.

Progress notions such as wait-freedom, lock-freedom, obstruction-freedom, and fair progress will likely require a different approach to represent in this framework.

Chapter 7

Performance Evaluation

We evaluated 4 different choices of the partition of variables. As we noted earlier, single writer variables can be safely removed from the partition without affecting correctness. In our experiments involving locks, we also safely removed the variables that were protected by the lock from the partition. We assume that the variables x that can be safely removed from the partition are indicated by the process in the predicate $\mathsf{safe}[x]$ and those that cannot by $\mathsf{unsafe}[x] = \neg\mathsf{safe}[x]$. The definitions of the models are in the summary of notation, Appendix A, or Section 3.1.

1. $K_S \stackrel{\text{def}}{=} \{\{x : x \in J\}\}$, $\mathsf{PartitionConsistency}(K_S) \equiv \mathsf{SequentialConsistency}$
2. $K_{SO} \stackrel{\text{def}}{=} \{\{x : x \in J \wedge \mathsf{unsafe}[x]\}\}$
3. $K_M \stackrel{\text{def}}{=} \{\{x\} : x \in J\}$, $\mathsf{PartitionConsistency}(K_M) \equiv \mathsf{PC\text{-}G}$
4. $K_{MO} \stackrel{\text{def}}{=} \{\{x\} : x \in J \wedge \mathsf{unsafe}[x]\}$

We evaluated these choices of partitions using the token and queue implementations described in Chapter 5. These choices resulted in a total of 8 distributed shared memory implementations. The 8 implementations are shown in the following table, where a shorthand is used to describe the composition of transformations applied to a program (P, J).

$$\Big(\mathrm{WRtrans}(K); \mathrm{XyPOB}\Big)\Big(P, J\Big) \stackrel{\text{def}}{=} \Big(\mathrm{XyPOB}\big(L(K)\big)\Big)\Big(\mathrm{WRtrans}(K)(P, J)\Big)$$

	unoptimized	optimized
Single token	SWFRtrans(K_S); TokenPOB	SWFRtrans(K_{SO}); TokenPOB
Multiple tokens	SWFRtrans(K_M); TokenPOB	SWFRtrans(K_{MO}); TokenPOB
Single queue	SWFRtrans(K_S); QueuePOB	SWFRtrans(K_{SO}); QueuePOB
Multiple queues	SWFRtrans(K_M); QueuePOB	SWFRtrans(K_{MO}); QueuePOB

Only the SWFRtrans implementation was tested as it was only realized later on that it was possible to implement FWSRtrans.

7.1 Experimental system specification

All experiments were performed on Westgrid's 128 node "matrix" cluster. Each node has a dual core 2.4 Ghz AMD Opteron processor, 2GB of RAM and runs Linux. The nodes are connected by a gigabit ethernet and a Voltaire Infiniband switched fabric interconnect. The network topology is a fat tree, and should provide consistent latency between nodes. The distributed shared memory implementations use the Message Passing Interface (MPI) API. The MPI implementation we use is HP-MPI [16], which takes advantage of the high speed interconnect.

All implementations were verified with a test to ensure that they properly supported mutual exclusion with the mutual exclusion protocol that is described in the next section. In the test, each processor acquired a lock and wrote its id to a large array. It then verified that all of the elements of the array were equal to its id before releasing the lock. If the lock failed because the memory consistency model provided by the implementation was too weak, then different ids would be read from the array.

verification-test()

```
    { Uses:
      – lock structure: lock
      – protected variables : safe ∀i ∈ 1, ..., large-array-length : large-array[i] = ⊥ }
1   for i ← 1 to test-repeats
2         do acquirelock(lock)
3            for j ← 1 to large-array-length
4                 do large-array[j] ← p
5            for j ← 1 to large-array-length
6                 do if large-array[j] ≠ p
7                       then report-error()
8            releaselock(lock)
```

7.2 Main experiment

We test a synthetic benchmark meant to mimic the pattern of lock usage of a real system. We use the unfair lock algorithm from Higham and Kawash [31]. Recall that we assume processes are numbered starting at 0.

Each lock has the following structure of read/write variables:

$$lock : \big(\textsf{unsafe}\ \ turn = \bot, \textsf{safe}\ \forall p \in P : flag[x] = \text{FALSE}\big)$$

The lock protocol is presented below, with p denoting the process that is executing the subroutine.

```
acquirelock(lock)
1   repeat
2            while ∃q > p : lock . flag[q]
3                 do if lock . flag[p]
4                       then lock . flag[p] ← FALSE
5            lock . flag[p] ← TRUE
6            lock . turn ← TRUE
7            while ∃q < p : lock . flag[q]
8                 do skip
             {in the until condition turn must be tested before flag  }
9       until (lock . turn = p) and (∀q > p : ¬ lock . flag[q])

releaselock(lock)
1   lock . flag[p] ← FALSE
```

The variables of this experiment are: (1) n the number of nodes used, (2) the DSM implementation used, and (3) m the number of locks available.

Each of the n nodes executes 300 critical sections on the chosen DSM implementation using the following code:

```
main-experiment()
    { Uses:
      − lock structures: ∀i ∈ 1, ..., m : lock[i]
      − protected variables : safe ∀i ∈ 1, ..., m : protected-variable[i] }
1   for i = 1 to 300
2         do random-lock ← rand({1, ..., m})
3            acquirelock(lock[random-lock])
4            for j = 1 to 5
5                 do write(protected-variable[random-lock], p)
6            releaselock(lock[random-lock])
```

The timing measurements were taken using MPI's Wtime function. Node 0 is assigned the task of recording the time at the beginning and end of the experiment. This measures the *total turnaround* time for the whole experiment. Beginning and end times are determined by barrier calls by all the processors at the start and the end of their tasks.

The hypothesis was that the increased concurrency allowed by the implementations with weaker memory consistency models would lead to increased performance.

The experiment was performed with all combinations of the following variable values:

1. The number of nodes: $|P| = 8, 16, 24$
2. The number of locks available: $m = 1, 2, 3, 4, 5, 6$
3. The DSM implementation (8 available)

Each experiment was performed 6 times to ensure a consistent result. The results are graphed in Figures 7.3, 7.4, and 7.5. The x-axis shows the number of critical

sections available to be randomly grabbed by processors. The y-axis shows the average total turnaround time in seconds for the experiment over the 6 trials, and the error bars show the maximum and minimum times.

The data did not support our hypothesis, but interesting patterns emerged in the data. With 8 nodes, the optimized single token implementation outperforms all of the others. This was surprising, as the optimized single token implementation allows less concurrency than the others. As the number of nodes increases to 16 and 24, the single token is no longer the best performer. A key factor in the performance of the token implementations is the *write-delay*. Each token is passed around a ring, so the time a node has to wait for a token grows linearly with the number of nodes. This factor explains why the token implementations suffer in performance relative to the queue implementations as performance increases.

It was expected that having multiple tokens would reduce the delay, as it would allow more tasks to be accomplished concurrently. However, the data shows that the multi-token implementations suffer from the additional overhead of managing the extra tokens. An explanation for this is that the task that requires a token, broadcasting a write and receiving its acknowledgments, is short enough that the added concurrency does not help. For the multi-token unoptimized implementation with 8 nodes, we see that performance actually decreases as locks are added. A congestion effect is responsible for this decrease. In the multi-token unoptimized implementation, there is a token for every variable. As the number of tokens increases, the system eventually has to spend more time managing tokens than managing the system and begins thrashing. For 16 and 24 nodes this thrashing does not occur since there are more nodes available to handle the tokens.

The unoptimized single and multiple queue implementations consistently perform well. These implementations perform nearly identically in the experiments. It was surprising to find, though, that the optimized queue implementations performed

consistently worse than the unoptimized queue versions. We decided to take more measurements to investigate this effect.

7.2.1 Investigating the number of writes

An initial explanation for the difference in performance was the mutual exclusion algorithm. Though every processor executed the same number of critical sections, differences in the memory consistency model and implementation could cause a difference in the number of retries the mutual exclusion section algorithm had to perform.

Since the number of writes not involved in the mutual exclusion algorithm is fixed, measuring the number of writes is a good way to see if the difference in implementations is affecting the execution of the mutual exclusion algorithm. We added code to the implementations to count the number of times the write routine was called.

The results of these measurements appear in Figures 7.6, 7.7, and 7.8. Apart from a few cases, the number of writes performed by each implementation seems relatively constant.

From this data we concluded that the number of lock retries was not the factor causing the difference between the implementations. The number of writes performed was not significantly different across implementations. We observed that the number of messages needed to perform each write could differ between implementations. This led us to measure the number of messages sent by the underlying protocols.

7.2.2 Investigating the number of sends

MPI offers a direct way to measure the number of messages sent by an MPI application by using its profiler. We performed the main experiment with the profiler to collect these measurements. The profiler significantly slowed down the performance of all the implementations, so we chose not to present the data since it was felt to be inaccurate.

We did use the data to lead us to the next experiment.

In the profiler runs, the optimized queue implementations were sending many more messages than the unoptimized queue implementations. But from the previous measurement, we knew that they were performing roughly the same number of writes.

By examining the code of the queue implementations, we can observe that the number of broadcast messages per write is fixed, but the number of acknowledgments can be very different. Attiya and Welch's protocol has an optimization that allows it to avoid sending an acknowledgment, it is in line 14 of HandleMessage() in Section 5.3. We instrumented the code to count the number of times the acknowledgment routine was called. Examples of the best-case and worst-case acknowledgment patterns of the queue algorithms are in Figures 7.1 and 7.2, respectively.

We ran the main experiments, measuring the number of update broadcasts that were performed. Since every update message is broadcast, we only record the number of broadcasts rather than the number of individual sends. The results are graphed in Figures 7.9, 7.10, and 7.11. We concluded from this data that updates were the reason for the reduced performance of the queue.

If this was caused by the pattern of writes, then it should be possible to reproduce this pattern without the mutual exclusion algorithm. We devised another benchmark to isolate the phenomenon.

7.3 Isolating update phenomenon

Rather than have the $|P|$ nodes attempt to acquire locks from a bank of m at random, we have m nodes perform the following pattern of writes:

mutex-removed-experiment()

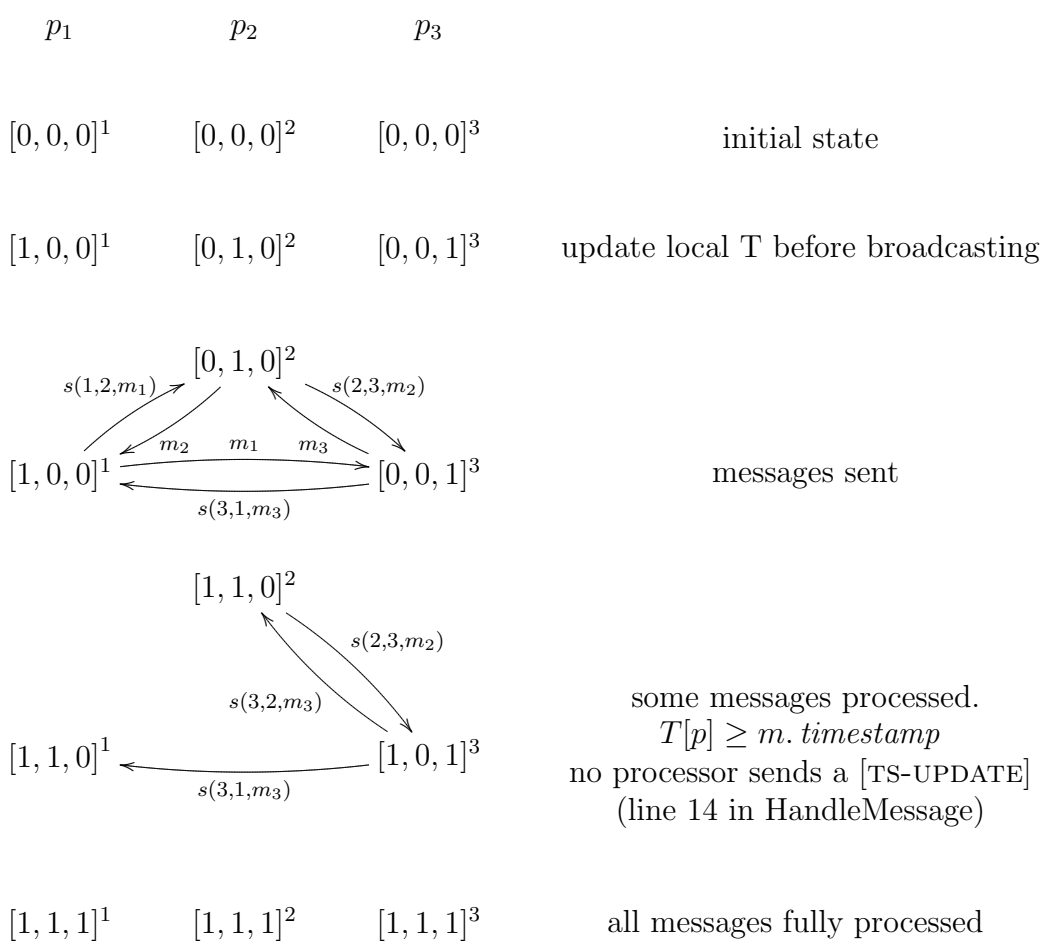


Figure 7.1: Example of best-case broadcast. Three messages broadcast using six messages.

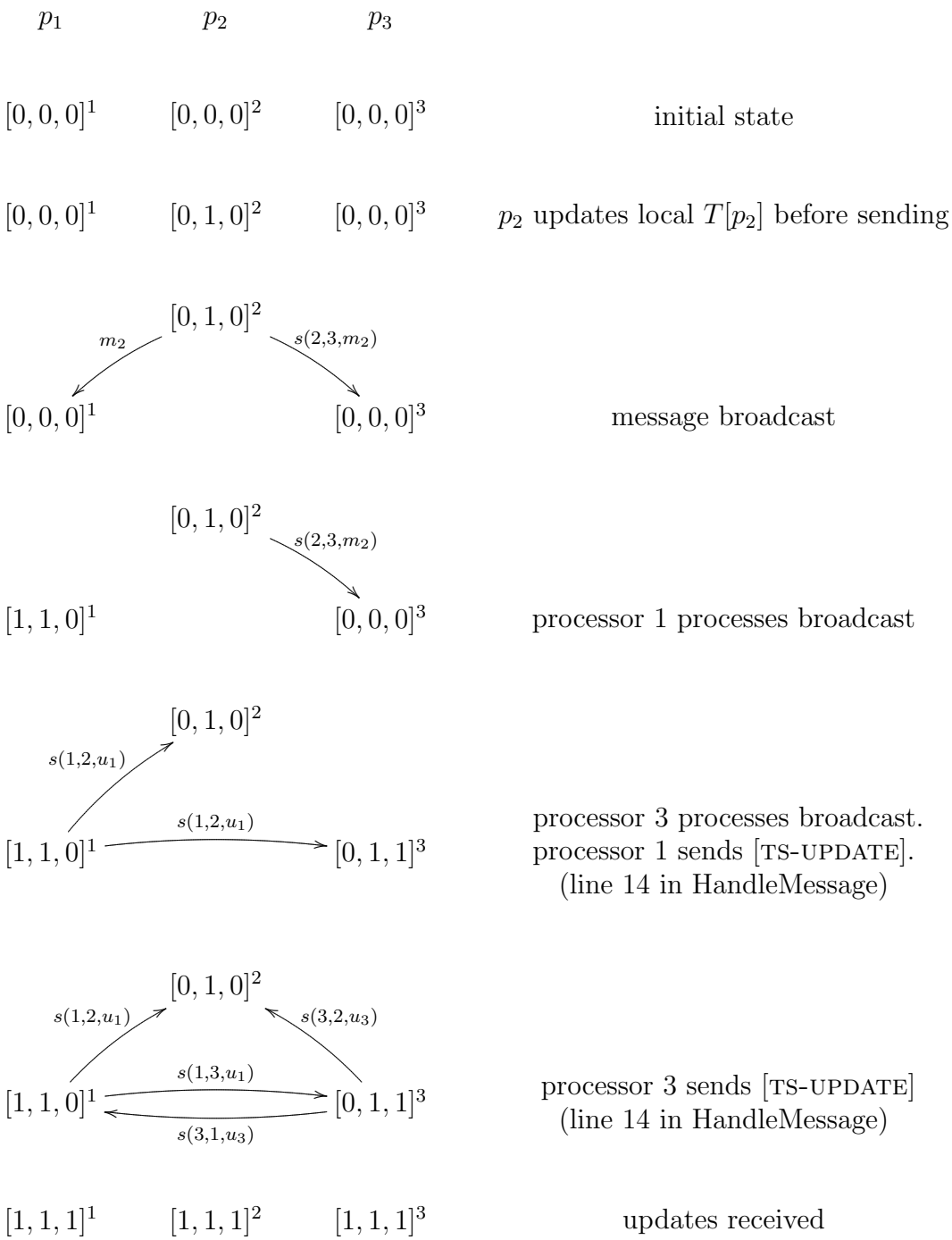


Figure 7.2: Example of worst-case broadcast. One message broadcast using six messages.

```
   { Uses:
     – fake lock variables : unsafe ∀i ∈ 1, ..., m : fake-lock-variable[i]
     – protected variables : safe ∀i ∈ 1, ..., m : protected-variable[i]  }
1  for i = 1 to 300×|P| / m
2        do write(fake-lock-variable[p], p)
3           for j = 1 to 5
4                 do write(protected-variable[p], p)
5           write(fake-lock-variable[p], p)
```

The premise behind the design is that in an execution of the primary experiment, $300 \times |P|$ lock acquires and releases are performed. For each acquire and release pair, the writes in lines 4-5 in main-experiment(), its *critical section* is performed. Since the locks are uniformly and randomly chosen from a bank of m, we would expect $\frac{300 \times |P|}{m}$ critical sections to be performed for each lock.

We wished to eliminate the effect of the mutual exclusion algorithm, but maintain the general pattern of writes that would be performed for each lock in main-experiment(). To achieve this we assigned each lock in main-experiment() a node perform all of the writes that would have been performed with that lock. Each of these m nodes performs mutex-removed-experiment(). Lines 3-4 in mutex-removed-experiment() are the same as the critical section, lines 4-5, of main-experiment(). We added one write at the beginning and the end of the critical section to an unsafe variable to represent the writes needed for lock entry and exit. A variation of mutex-removed-experiment() was performed where the number of loops in line 3 was varied from 1 to 5, with similar results.

So despite the fact that the optimized queue algorithms could perform fast writes with 0 message delay, this threw off the synchronization of the system, causing more acknowledgments to be sent. This factor caused the implementations to slow down significantly.

7.4 Conclusions of performance evaluation

Most of the implementations failed to improve on performance of the single queue implementations. Multiple tokens and multiple queues weakened the consistency and increased concurrency, but did not provide any performance gain.

The optimized single token implementation outperformed all of the other implementations with 8 nodes. Relaxed consistency allowed us to maintain the simplicity of the implementation, while increasing performance. The mutual exclusion algorithm depends on one unsafe variable, which requires a token when writing to it. This unsafe variable can then be used to protect many other variables that are safe and do not depend on the token. This feature allowed the writes inside the critical section to complete faster and with less overhead in the optimized implementation. More work is required to investigate this implementation, $\big(\text{SWFRtrans}(K_{SO}); \text{TokenPOB}\big)$, and its associated model $\mathsf{PartitionConsistency}(K_{SO})$.

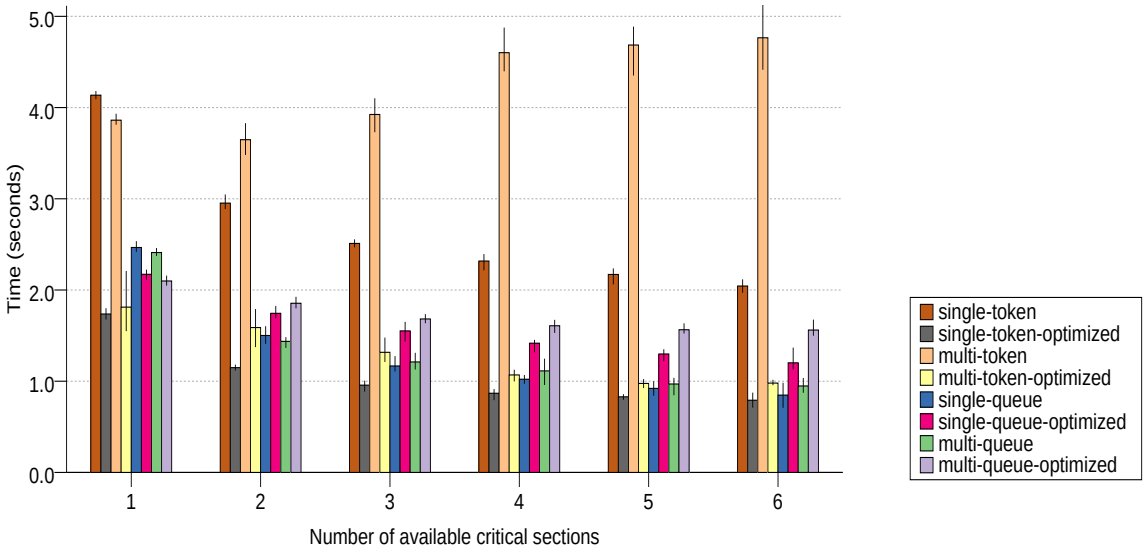


Figure 7.3: Total turnaround time for 8 processors, each performing 300 critical sections. Single token with optimization is the fastest. For both queue implementations, the optimized versions are slower.

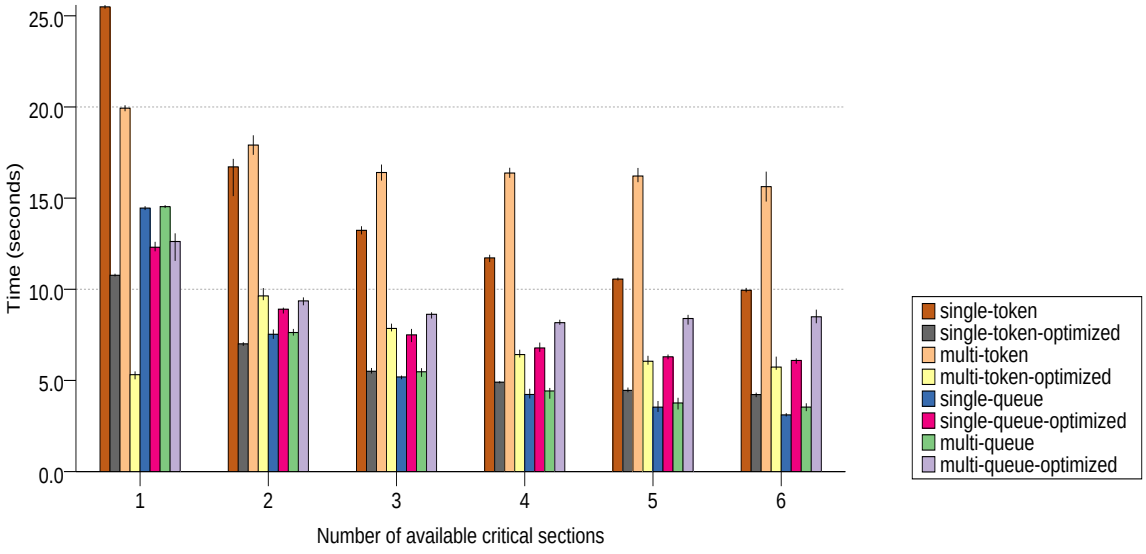


Figure 7.4: Total turnaround time for 16 processors, each performing 300 critical sections. Single queue is fastest implementation. Optimized queue implementations slower than regular queue implementations.

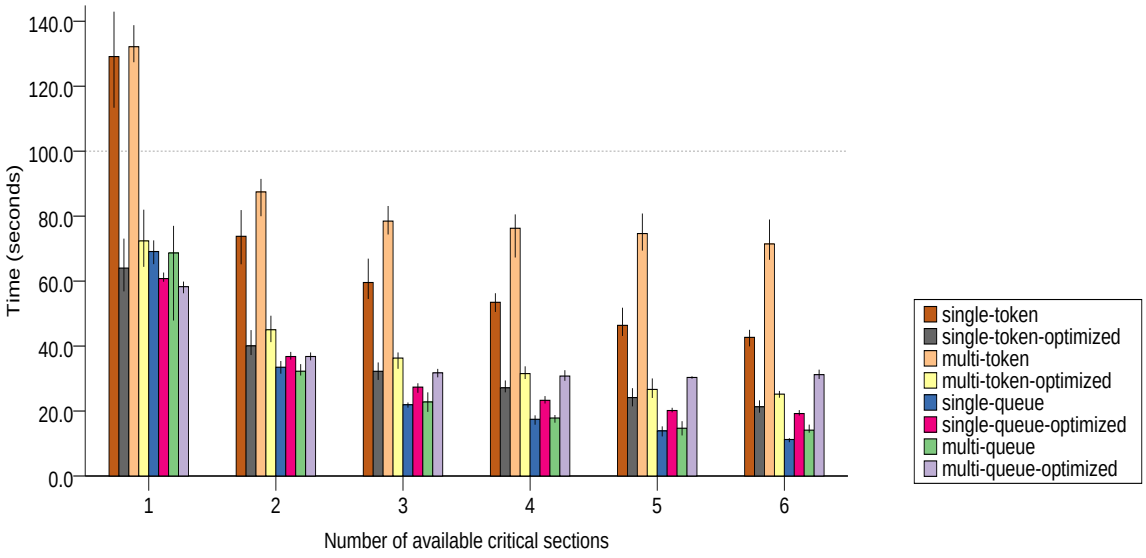


Figure 7.5: Total turnaround time for 24 processors, each performing 300 critical sections. Single queue is fastest implementation. Optimized queue implementations slower than regular queue implementations.

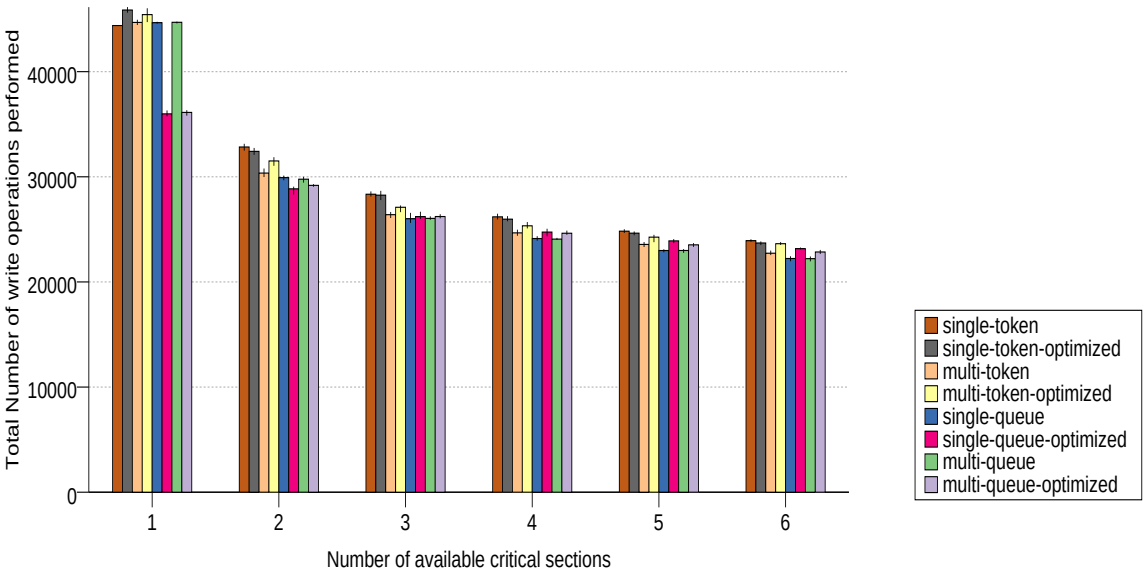


Figure 7.6: Total number of writes for 8 processors, each performing 300 critical sections.

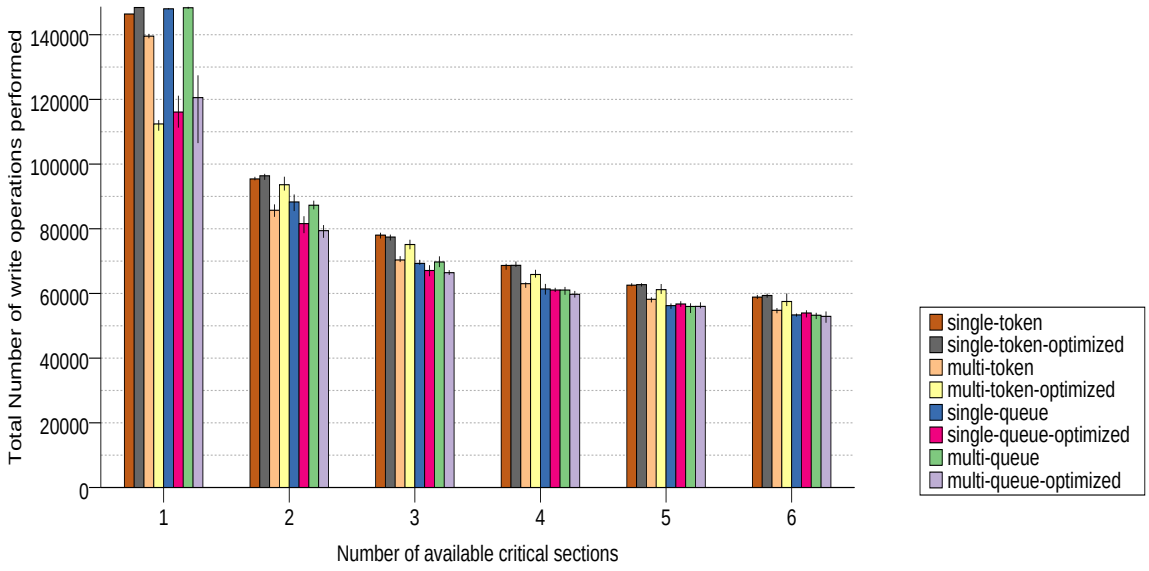


Figure 7.7: Total number of writes for 16 processors, each performing 300 critical sections.

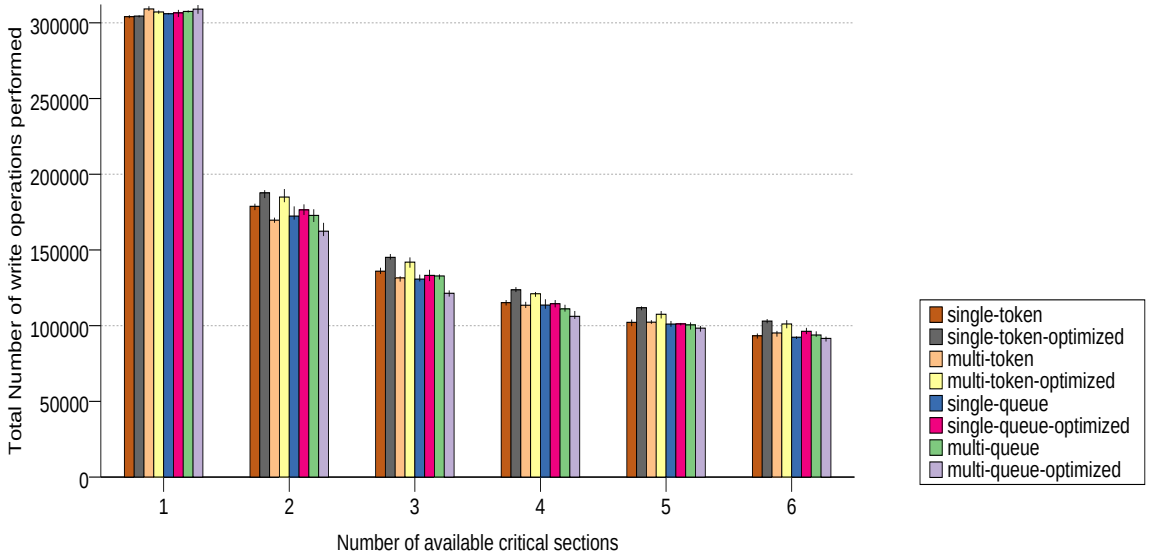


Figure 7.8: Total number of writes for 24 processors, each performing 300 critical sections.

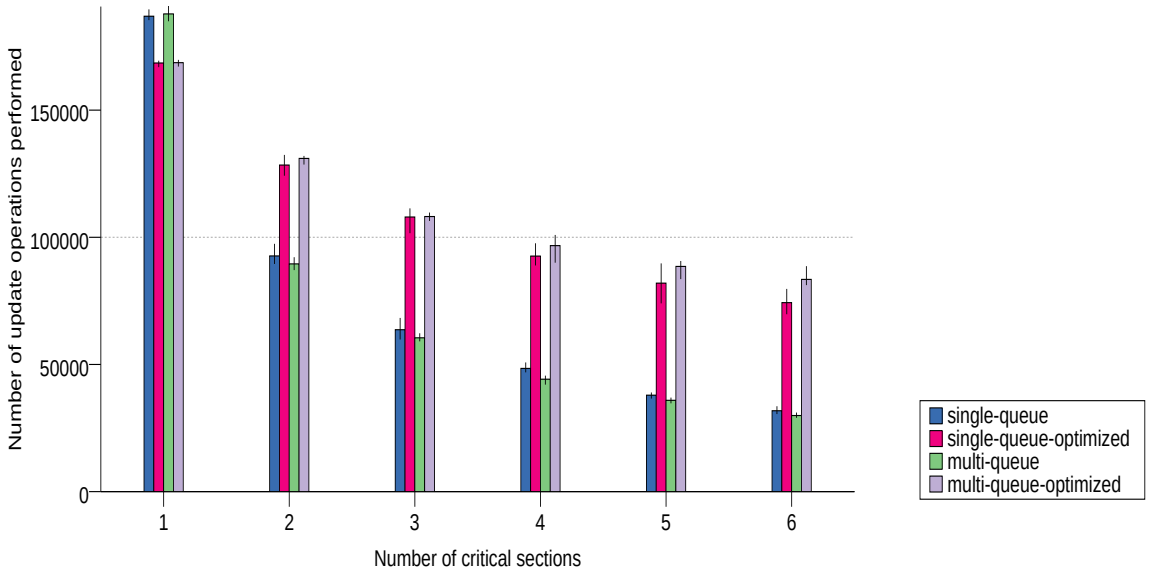


Figure 7.9: Total number of update operations for 8 processors, each performing 300 critical sections

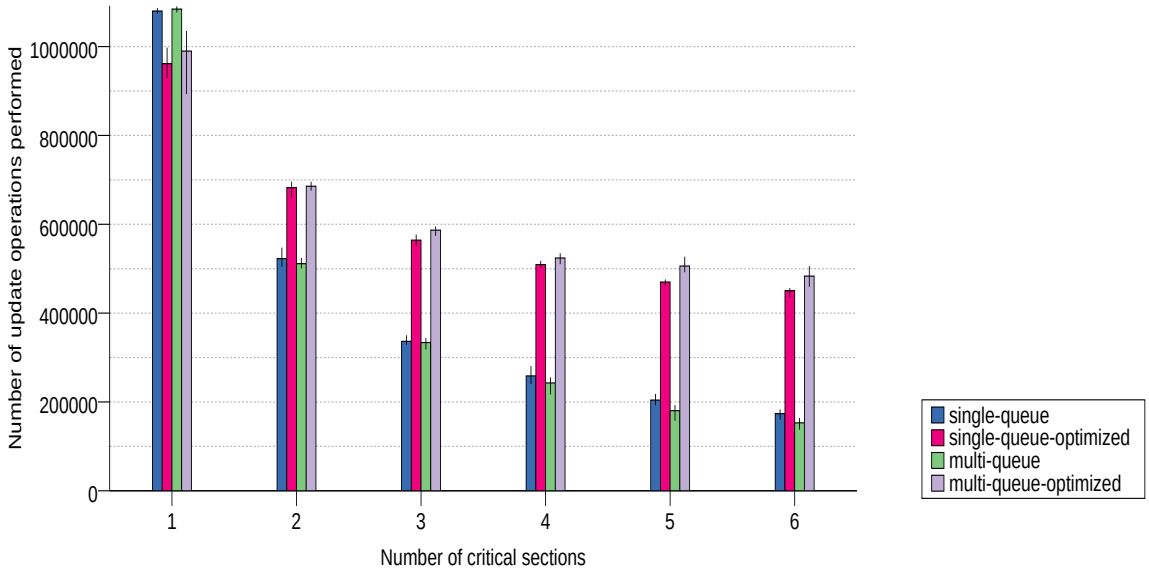


Figure 7.10: Total number of update operations for 16 processors, each performing 300 critical sections

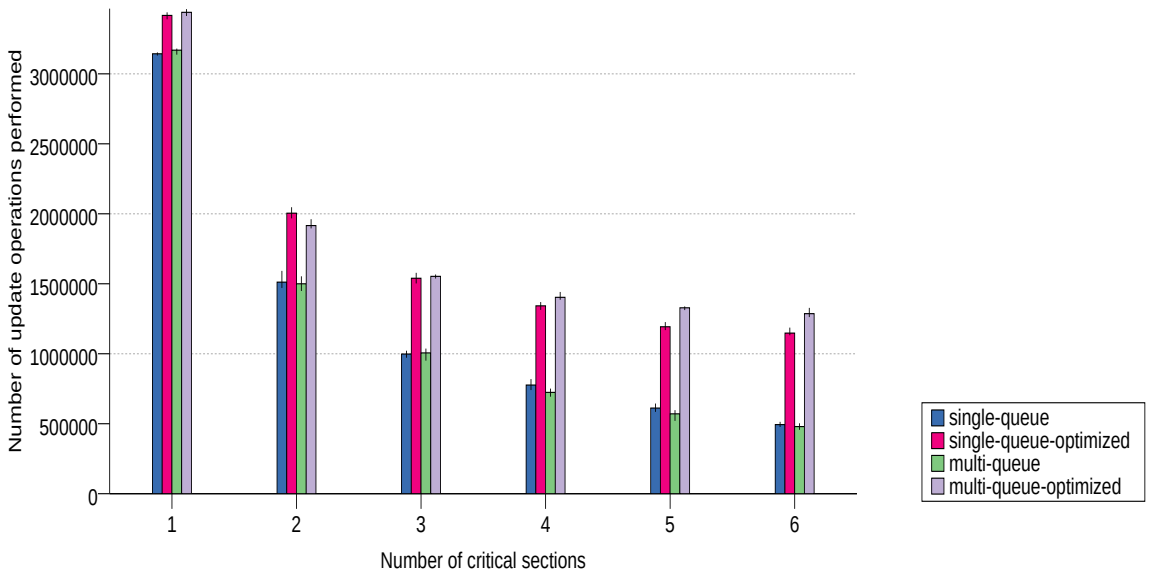


Figure 7.11: Total number of update operations for 24 processors, each performing 300 critical sections

Chapter 8

Conclusion

8.1 Summary

This thesis introduced several new distributed shared memory implementations. To prove these implementations correct, we modeled the implementation and specification platforms using memory consistency modeling techniques.

To model the platforms that we implement on, we use memory consistency techniques to specify a network of multiprocessors. To generically model our specification platforms, we introduce the definition of a new class of platforms, Partition Consistency. The Partition Consistency class of platforms includes Pipelined-RAM, PC-G, and Sequential Consistency as special cases. Finally, we introduce an intermediate platform that allows us to separate the broadcasting mechanism from the replica management, making our proofs clearer and more structured.

The Partition Consistency model is a generalization of previously defined models, based mostly on the PC-G definition in [4]. The network model integrates Lamport's notion of happens-before and causality with sequential consistency. The innovation here is bringing these concepts together in a memory consistency model. We generalize the notion of total order broadcast, to partial order broadcast, a new definition. Modelling a broadcast based system with a memory consistency model is also new.

To this modeling we added the concept of program transformation to model the mechanism for implementing one platform on another. We introduce a general principle for proving that program transformations are implementations, based on previous work on implementing memory consistency models. This principle is flexible enough to allow the introduction of threads and message passing.

The proof techniques used in this thesis are based on a large body of previous work by Higham, Jackson, Kawash, and Verwaal. This thesis introduces further formalization of these techniques, and refines them so that they now also apply to transformations that add threads to implementations. This was done to make use of the new models introduced.

We used the general principles together with the models to define our implementations and prove them correct. We proved the correctness of four implementations, two similar implementations that manage replication and two different implementations that manage broadcasting, one using tokens and one using timestamps and priority queues. To provide these proofs in a more accessible and detailed format, we introduce a proof diagram format.

The implementations are based on implementation techniques found in the literature. Token rings are a classic technique in distributed systems, and Attiya and Welch used timestamps and priority queues to implement totally ordered broadcast. We used these techniques to implement Partition Consistency, and therefore, PC-G as a special case. PC-G has not been implemented before, making this application of the techniques novel.

Based on the formal implementations of Partition Consistency that were proved correct, we created distributed shared memory implementations of Partition Consistency platforms using C++ and MPI. These implementations were used to run performance experiments, comparing performance between the implementations. A few unexpected results came out of the performance evaluations, which were investigated. The results of the performance evaluations provide many insights into the performance characteristics of the implementations.

8.2 Future directions

In all of the areas that this thesis involves, there is considerable room for future work.

The performance experiments we performed had interesting and unexpected results. It was clear that many different factors, including message congestion, timing, and protocol design could have a significant impact on performance. In-depth analysis is often needed to discover these factors and they are difficult to isolate. Discovering general principles of DSM design and their effect on performance would be useful to investigate in academic, open source, and commercial shared memory implementations.

There remain many challenges that require further development of memory consistency modeling techniques. With recent interest in highly concurrent data structures, it has become clear that some programming languages need to expose the underlying memory model of the platform. To handle this portably, languages can create their own memory model that compilers maintain by adding appropriate strong memory operations. Memory consistency models that integrate programming language semantics and more general program transformations would allow programming language compilers and interpreters to be modeled. General techniques are also needed to model common microprocessor optimizations such as pipelining and instruction reordering.

Progress is an important factor that is missing from memory consistency models. Memory consistency models abstract the implementation details of the underlying memory system, but at the same time, hide its progress characteristics. For example, we may require that two write operations are seen in a certain orders, but we need a separate progress requirement to ensure that they will eventually be seen.

Memory consistency modeling techniques extend far beyond memory consistency. A recent paper [49] by Spear, Dalessandro, Marathe and Scott uses very similar modeling techniques, which they call, *ordering-based semantics* to model software transactional

memory. Weikum and Vossen provide similar ordering based definitions of transactional systems in Transactional Information Systems [52].

Large distributed systems often use some form of caching, however it is often used very conservatively because it is difficult to synchronize the caches. This is similar to the management of replicas in this thesis. Order based modeling techniques could be used to better describe and reason about the design of these caching systems.

Many large distributed systems require fault tolerance mechanisms because a large number of components increases the probability that some component will fail. Failures that simply cause components to stop would be handled by an integration of order based modeling and progress conditions. Failures that cause components to communicate incorrect results have not to our knowledge been modeled using these techniques.

The proof method we presented should be extended to deal with implementation of concurrent data structures and transaction models. Both the proofs and the models rely heavily on relations. It may be possible to obtain a more general and algebraic theory by using the theory of allegories, mathematical structures that abstract the notion of relations on sets. Brown and Hutton provide [13] references to allegory theory, as well as their application to the design of circuits. Brown and Hutton's work also develops an algebra of picture diagrams based in allegory theory, this may indicate a similar way to represent our proof diagrams with allegories. Possibilities of improving the proof method and techniques to deal with the modeling situations described would include proofs of implementations of highly concurrent data structures, transactional memory, transactional databases, file systems, and programming language compilers.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] Luca Aceto, Anna Inglfsdttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 8 2007.

[3] Divyakant Agrawal, Manhoi Choy, Hong Va Leong, and Ambuj K. Singh. Investigating weak memories using maya. In *Proceedings of the 3rd International Symposium on High Performance Distributed Computing*, pages 123–130, 1994.

[4] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the 5th International Symposium on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology.

[5] David Aspinall and Jaroslav Sevcík. Formalising java's data race free guarantee. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, pages 22–37, 2007.

[6] Hagit Attiya and Roy Friedman. Programming DEC-Alpha based multiprocessors the easy way. In *Proceedings of the 6th International Symposium on Parallel Algorithms and Architectures*, pages 157–166, 1994. Technical Report LPCR 9411, Computer Science Department, Technion.

[7] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[8] Hagit Attiya and Jennifer L. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2004.

[9] Ralph-Johan Back, Jim Grundy, and Joakim von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9(5-6):469–483, 1997.

[10] Roland Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley and Sons, Inc., New York, NY, USA, 2003.

[11] Michael Barr and Charles Wells. *Category Theory for Computing Science, third edition*. Les Publications CRM, Montréal, 1999.

[12] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.

[13] Carolyn Brown and Graham Hutton. Categories, allegories and circuit design. In *LICS*, pages 372–381. IEEE Computer Society, 1994.

[14] Vicent Cholvi, Antonio Fernández, Ernesto Jiménez, and Michel Raynal. A methodological construction of an efficient sequential consistency protocol. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications*, pages 141–148. IEEE Computer Society, 2004.

[15] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.

[16] Hewlett Packard Development Company. *HP MPI User's Guide*. 2003. Order number: B6060-96022.

[17] Francisco Corella, Janice M. Stone, and Charles Barton. A formal specification of the PowerPC shared memory architecture. Technical Report RC18638, IBM, 1994.

[18] Intel Corporation. Intel 64 memory ordering white paper. Technical Report SKU:318147-001, Intel Corporation, 2007.

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A: System Programming Guide, Part 1. 2008. SKU:253668.

[20] Compaq Computer Corportaion. *The Alpha Architecture Handbook*. 1998. Order number: EC-QD2KC-TE.

[21] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[22] W. H. J. Feijen and A. J. M. van Gasteren. *On a method of multiprogramming*. Springer-Verlag New York, Inc., New York, NY, USA, 1999.

[23] Rob Gerth. Sequential consistency and the lazy caching algorithm. *Distributed Computing*, 12(2-3):57–59, 1999.

[24] Jay L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61:199–224, 1988.

[25] James Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[26] David Gries and Fred. B. Schneider. *A Logical Approach to Discrete Math (Monographs in Computer Science)*. Springer, 1st edition, October 1993.

[27] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[28] Lisa Higham, LillAnne Jackson, and Jalal Kawash. Capturing register and control dependence in memory consistency models with applications to the Itanium architecture. In *Proceedings of the 20th International Symposium on Distributed Computing*, September 2006.

[29] Lisa Higham, LillAnne Jackson, and Jalal Kawash. Programmer-centric conditions for itanium memory consistency. In *Proceedings of the 8th International Conference on Distributed Computing and Networking*, December 2006.

[30] Lisa Higham, LillAnne Jackson, and Jalal Kawash. Specifying memory consistency of write buffer multiprocessors. *ACM Transactions on Computer Systems*, 25(1), 2007.

[31] Lisa Higham and Jalal Kawash. Tight bounds for critical sections in processor consistent platforms. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1072–1083, 2006.

[32] Lisa Higham and Jalal Kawash. Implementing sequentially consistent programs on processor consistent platforms. *Journal of Parallel and Distributed Computing*, 68(4):488–500, April 2008.

[33] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing Systems*, pages 349–356, October 1997.

[34] C.A.R. Hoare. *Communicating Sequential Processes (Prentice-Hall International Series in Computer Science)*. Prentice Hall, April 1985.

[35] Javid Huseynov. Distributed shared memory home pages. http://www.ics.uci.edu/∼javid/dsm.html.

[36] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter*, pages 115–132, 1994.

[37] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[38] Leslie Lamport. On interprocess communication (parts I and II). *Distributed Computing*, 1(2):77–85 and 86–101, 1986.

[39] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, July 2002.

[40] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.

[41] Nancy A. Lynch. *Distributed Algorithms (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 1st edition, April 1997.

[42] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[43] Robin Milner. *Communication and Concurrency (Prentice Hall International Series in Computer Science)*. Prentice Hall, 1st edition, September 1995.

[44] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.

[45] Olaf Müller. I/o automata and beyond: Temporal logic and abstraction in isabelle. In Jim Grundy and Malcolm C. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 331–348. Springer, September 1998.

[46] Vaughan R. Pratt. Modeling concurrency with geometry. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 311–322, January 1991.

[47] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[48] Cheng Shao, Evelyn Pierce, and Jennifer Welch. Multi-writer consistency conditions for shared memory objects. In *Distributed algorithms*, volume 2848/2003 of *Lecture Notes in Computer Science*, pages 106–120, Oct 2003.

[49] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. Ordering-based semantics for software transactional memory. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, volume 5401 of *Lecture Notes in Computer Science*, pages 275–294. Springer, December 2008.

[50] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, 2004.

[51] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–136. ACM, March 2006.

[52] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 1st edition, May 2001.

Appendix A

Summary of Notation and Definitions

A.1 General math notation

Notation	**Meaning**	**Example**
$\mathsf{pred}[x_1, x_2, \ldots, x_n]$	n-place predicate	$\mathsf{divides}[x, y]$
$\langle\, x_a : a \in A : \mathsf{pred}[a, \ldots]\,\rangle$	collection of x_a, one for each $a \in A$	$\langle\, f_a : a \in A : f_a = g(a)\,\rangle$
$\underset{R}{\longrightarrow}$	R is a relation on the operations O_C of a computation C, often a partial order	$\underset{\text{HappensBefore}}{\longrightarrow}$
$\underset{R}{\models}$	R is an interpretation relation between computations of two systems	$\underset{\text{SWFRtrans}^{*-1}}{\models}$
$\underset{R}{\sim}$	R is a constructed relation between specification and target level operations	$\underset{construct}{\sim}$
$\mathsf{Extends}[\underset{R}{\longrightarrow}, \underset{T}{\longrightarrow}, A]$	$\forall a_1, a_2 \in A : a_1 \underset{T}{\longrightarrow} a_2 \implies a_1 \underset{R}{\longrightarrow} a_2$	
$\mathsf{Agree}[\rightarrow_R, \rightarrow_T, A]$	$\forall a_1, a_2 \in A : (a_1 \underset{R}{\longrightarrow} a_2) \Leftrightarrow (a_1 \underset{T}{\longrightarrow} a_2)$	

A.2 Memory consistency notation and convention

Notation	**Meaning**
P	multiprocess
J	set of program objects
(P, J)	multiprogram
(P, J, MC)	system, where MC is a memory consistency model
C	computation
$\widehat{C}$	target level computation
p	process
O_C	set of operations in computation C, subscript omitted when C clear from context
$\mathcal{C}(P, J)$	computations generated by program (P, J)
$\mathcal{C}(P, J, \mathsf{MC})$	computations generated by system (P, J, MC)
$\tau^*(C)$	trace transformation of a computation C. τ^* is derived from a program transformation τ.
$\models_{\tau^{*-1}}$	inverse of the trace transformation τ^* derived from a program transformation τ

A.3 Proof diagrams

Notation	**Meaning**
$a = b$	asserts that $a = b$
$a \xrightarrow[L]{} b$	asserts that $a \underset{L}{\longrightarrow} b$.
$\begin{array}{ccc} a & \xrightarrow{L} & b \\ & \Downarrow & \\ c & \xrightarrow[M]{} & d \end{array}$	asserts that $(a \underset{L}{\longrightarrow} b) \implies (c \underset{M}{\longrightarrow} d)$
$A \xrightarrow[L]{} B$	where A and B are sets asserts that $\forall a \in A, b \in B$: $a \underset{L}{\longrightarrow} b$.
$A \vdash \xrightarrow[L]{} B$	where A and B are sets asserts that $\exists b \in B : \forall a \in A$: $a \underset{L}{\longrightarrow} b$.
$A - \xrightarrow[L]{} \nrightarrow B$	where A and B are sets asserts that $\exists a \in A : \forall b \in B$: $a \underset{L}{\longrightarrow} b$.
$A \subset B$	asserts that $A \subset B$.
$A \subseteq B$	asserts that $A \subseteq B$.
$a - \overline{R} - b$	asserts that a is related to b by a relation R.

A.4 Memory consistency models

A.4.1 Partition Consistency

$$\mathsf{PartitionConsistency}(K)[C] \stackrel{\text{def}}{=}$$
$$[\mathbf{VTO}]\exists \langle\ (O|p \cup O|\mathsf{writes}, \underset{L_p}{\longrightarrow}) \text{ valid total order} : p \in P$$
$$: \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O|p \cup O|\mathsf{writes}]\ \rangle :$$
$$[\mathbf{AGR}]\forall p, q \in P, i \in [1,k] : \mathsf{Agree}[\underset{L_p}{\longrightarrow}, \underset{L_q}{\longrightarrow}, O|\mathsf{writes}(S_i)]$$

A.4.2 Partial order broadcast

$$\mathsf{POB\text{-}Cluster}(L)[C] \stackrel{\text{def}}{=}$$
$$[\mathbf{VTO}]\exists \langle\ (O|p, \underset{L_p}{\longrightarrow}) \text{valid total orders} : p \in P : \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow} \cup \underset{delorder}{\longrightarrow}, O|p]\ \rangle :$$
$$[\mathbf{AGR}]\forall p, q \in P : (\text{label}(m_1) = \text{label}(m_2)$$
$$\wedge \frac{\texttt{deliver()}}{m_1} \underset{L_p}{\longrightarrow} \frac{\texttt{deliver()}}{m_2}$$
$$\wedge \frac{\texttt{deliver()}}{m_2} \in O|q$$
$$) \implies \frac{\texttt{deliver()}}{m_1} \underset{L_q}{\longrightarrow} \frac{\texttt{deliver()}}{m_2}$$

We have an alternate definition that explicitly arranges all the properties that need to be proved.

$$\mathsf{POB\text{-}Cluster}(L)[C] \stackrel{\text{def}}{=}$$
$$\exists\, (M, \underset{MO}{\longrightarrow}) \text{ partial order }, \langle\ (O|p, \underset{L_p}{\longrightarrow}) : p \in P :\ \rangle :$$
$$\left(\frac{\texttt{deliver}()}{m} \in O \implies m \in M\right)$$
$$\wedge \big(\text{a message is delivered only once per processor}\big)$$
$$\wedge \left(\forall l \in L : (M|l, \underset{MO}{\longrightarrow}) \text{ is total }\right)$$
$$\wedge \left(\texttt{bcast}(m_1) \underset{\text{prog}}{\longrightarrow} \texttt{bcast}(m_2) \implies m_1 \underset{MO}{\longrightarrow} m_2\right)$$
$$\wedge \left(\texttt{bcast}(m), \frac{\texttt{deliver}()}{m} \in O|p \implies \texttt{bcast}(m) \underset{L_p}{\longrightarrow} \frac{\texttt{deliver}()}{m}\right)$$
$$\wedge \left(m_1 \underset{MO}{\longrightarrow} m_2 \wedge \frac{\texttt{deliver}()}{m_2} \in O|p \implies \frac{\texttt{deliver}()}{m_1} \underset{L_p}{\longrightarrow} \frac{\texttt{deliver}()}{m_2}\right)$$
$$\wedge \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{prog}}{\longrightarrow}, O|p]$$
$$\wedge \left((O|p, \underset{L_p}{\longrightarrow}) \text{ is a valid total order }\right)$$

A.4.3 Threaded Network

$$x \underset{\text{MessageOrder}_C}{\longrightarrow} y \stackrel{\text{def}}{=} x, y \in O_C \wedge x = \texttt{send}(s, d, m)\ \wedge y = \frac{\texttt{recv}()}{s, d, m}$$
$$x \underset{\text{WritesInto}_C}{\longrightarrow} y \stackrel{\text{def}}{=} x, y \in O_C \wedge x = \texttt{write}(w, z) \wedge y = \frac{\texttt{read}(w)}{z}$$
$$x \underset{\text{FifoChannel}_C}{\longrightarrow} y \stackrel{\text{def}}{=} x, y \in O_C \wedge \frac{\texttt{recv}()}{s, d, m} \wedge y = \frac{\texttt{recv}()}{s, d, m'} \wedge \texttt{send}(s, d, m) \underset{\text{prog}}{\longrightarrow} \texttt{send}(s, d, m')$$
$$\underset{\text{HappensBefore}_C}{\longrightarrow} \stackrel{\text{def}}{=} (\underset{\text{MessageOrder}}{\longrightarrow} \cup \underset{\text{prog}}{\longrightarrow} \cup \underset{\text{WritesInto}}{\longrightarrow} \cup \underset{\widehat{\text{FifoChannel}}}{\longrightarrow})^+$$

$$\mathsf{ThreadedNetwork}[C] \stackrel{\text{def}}{=}$$
$$[\mathbf{VTO}] \exists\ \langle\ (O|p, \underset{L_p}{\longrightarrow}) \text{ valid total order }\ : p \in P : \mathsf{Extends}[\underset{L_p}{\longrightarrow}, \underset{\text{HappensBefore}}{\longrightarrow}, O|p]\ \rangle$$