

Partition Consistency: A Class of Memory Consistency Models with Implementations on Message-Passing Networks

Steven Cheng, Lisa Higham, Jalal Kawash
University of Calgary
{hyswchen,higham,jkawash}@ucalgary.ca

Abstract

Partition Consistency is a new class of memory consistency models, which unifies and generalizes many known consistency models. For example, sequential consistency, pipelined-RAM, and processor consistency are specific instantiations of the Partition Consistency class. Four different implementations of the general Partition Consistency Class on message-passing networks with multi-threaded nodes are developed and proved correct. These implementations and their proofs of correctness are facilitated by using a middle level of abstraction, which generalizes a total order broadcast model to a partial order one. The same framework is used to specify the memory consistency model of each of the three levels of abstraction. This simplifies the proofs and supports the composition of the implementations from level to level. We briefly summarize some performance studies of two of our four possible implementations when applied to four specific instantiations: Sequential Consistency, processor consistency, and a weakened version of each of these. These studies uncovered a new model, weak Sequential Consistency, which appears to offer some efficiency gains though further study is needed.

Keywords: memory consistency models, (Weak) Sequential Consistency, Processor Consistency, partial-order broadcast, distributed-shared memory

1 Introduction

Partition Consistency defines a new general class of memory consistency models, which captures different degrees of consistency between various sets of shared variables, as is common in many distributed and multiprocessor environments. Software examples such as Itanium[13] and Java[18] also exhibit differing degrees of consistency depending on how variables are declared. Abstract consistency models such as Sequential Consistency (SC) [15], pipelined-RAM [17], and processor consistency (PC-G) [1], are special cases of Partition Consistency.

Generally, the weaker the memory consistency requirements of a multiprocessing model, the more efficiently the model can be implemented, but the more difficult it is to program correctly. By exploring various models and understanding how they are related, we gain insight into how and when to use them, and expose general techniques for implementing them on lower level models of multiprocessors or networks. Our goal is to implement any instance of the Partition Consistency Class on a message-passing network with multi-threaded nodes, with a view to understanding the possible efficiencies and at what programming cost. Our implementations and their proofs of correctness are facilitated by a middle level of abstraction, which generalizes a total order broadcast model to a partial order one. Thus we use three levels of abstraction: the abstract Partition Consistency Model, the intermediate Partial Order Broadcast Model, and the concrete Message-Passing Network Model. The same framework is used to define the memory consistency model of each level. So the intermediate Partial Order Broadcast Model is first the *target* of the implementation of the *specified* Partition Consistency Model. Next the same Partial Order Broadcast abstraction serves as the specified model, which is implemented on the target Message-Passing Network Model. We define both a fast-read/slow-write and a fast-write/slow-read implementation of Partition Consistency on Partial Order Broadcast. Next we give both a token-based and a queue-based implementation of Partial Order Broadcast on Message-Passing Networks. This results in four compositions of transformations; each implements Partition Consistency on Message-Passing Networks.

Proofs of correctness of implementations of shared memory models on multiprocessors or networks typically involve a great deal of tedious but essential detail, and are thus prone to imprecision and error. Layering the implementation on levels that are defined using a common framework help overcome these problems. by allowing us to focus on only part of the proof obligation at each level. We also introduce a diagrammatic notation for our proofs. These diagrams provide a visual representation of logical statements and inferences. We think they contribute precision and conciseness and eliminate some possible ambiguities of verbal proofs.

To evaluate performance, we assessed the two slow-write/fast-read implementations of each of four specific instantiations of Partition Consistency: Sequential Consistency, Processor Consistency, and a weakened version of each of these. Our experiments show that the increased concurrency allowed by weaker memory consistency models does not necessarily translate to better performance. Our studies, however, uncovered a new model, Weak Sequential Consistency, which appears to have the potential to outperform both SC and PC-G.

Related work: A lot of related research influenced this work; only the most closely related are discussed next. This research benefited from Lamport's techniques for establishing correctness of distributed systems [16] and Herlihy and Wing's sequential specification of objects [8]. Vafeiadis *et. al.* use similar modeling techniques to prove the correctness of concurrent data structures, but only in linearizable settings [19]. Higham *et. al.* develop a general method of dealing with weak memory models and apply it to SPARC and Itanium multiprocessors [10, 11]. Aspinal *et. al.* formalize the Java Memory Model (JMM) using a style similar to ours, and provide machine verified proofs [2]. Cholvi *et. al.* present a DSM protocol that is limited to SC [6]. Our implementations of Partition Consistency on the Partial Order Broadcast model generalize Attiya and Welch's implementation of Sequential Consistency using total order broadcast [3, 4].

Organization of the rest of the paper: Section 2 provides the three levels of abstraction used in this

check for accuracy

more precise what
Cholvi did and how it
relates

paper, by defining the memory consistency of each level. Section 3 describes the implementations. Section 4 presents pieces of the proofs. Space constraints force us to omit parts, summarize much, and only sketch some proofs, elaborating further in the Appendix. The full version of the paper [5] contains all the details.

2 Definitions and Models

A *multiprogram* is a collection of individual computer processes accessing a collection of shared objects. Each process in the multiprogram consists of one or more *threads*; each thread generates *operation invocations* in a sequence called *program order* that is consistent with its program. A *computation* of the multiprogram is formed by *arbitrarily* completing each operation invocation, in each individual thread sequence, with a response, creating a collection of sequences of (*completed*) *operations* (also called program order) — one sequence for each thread of each process. We denote the unrestricted set of computations of a multiprogram P by $\mathcal{C}(P)$. The subset of $\mathcal{C}(P)$ that could actually result from the execution of the multiprogram depends upon the multiprocessor architecture. A *memory consistency model* is a predicate defined on the set of all possible computations of a multiprogram; it filters these computations to include only those that could arise on the architecture being modelled. The subset of $\mathcal{C}(P)$ that satisfies the memory consistency predicate, MC, is denoted $\mathcal{C}(P, \text{MC})$.

We use the following notation, terminology and conventions for the rest of the paper. For a computation C of a multiprogram P , O_C (omitting the subscript C when it is obvious) denotes all the operations of C . A completed operation with input u that returns a value v is denoted $\frac{\text{OPER}(u)}{v}$. $O|_{\text{writes}}(S)$ denotes the set of all write operations in O to variables in S ; if S is all the variables, we write $O|_{\text{writes}}$. $O|_p$ denotes the set of all operations by process $p \in P$. The *program order relation on O_C* , denoted $\xrightarrow{\text{prog}}$, is the partial order formed by the union of the individual thread program orders¹. The style $\text{pred}[\text{args}]$ is used to denote a predicate. Given relations \xrightarrow{R} , \xrightarrow{T} , and a set A , define:

extension of a relation : $\text{Extends}[A, \xrightarrow{R}, \xrightarrow{T}] \stackrel{\text{def}}{=} \forall a_1, a_2 \in A : a_1 \xrightarrow{T} a_2 \implies a_1 \xrightarrow{R} a_2$
 agreement of relations: $\text{Agree}[A, \xrightarrow{R}, \xrightarrow{T}] \stackrel{\text{def}}{=} \forall a_1, a_2 \in A : (a_1 \xrightarrow{R} a_2) \iff (a_1 \xrightarrow{T} a_2)$.

For a finite set A , we overload the term *total order*: it refers to either the set of ordered pairs (A, \xrightarrow{R}) in the order, or the sequence, R , that realizes that total order. The notation $\langle x_a : a \in A \rangle$ specifies a collection of items x_a , exactly one for each $a \in A$.

The most common shared objects for this paper are *variables* with the sequential specification [9]: a sequence of READ and WRITE operations on a variable x is *valid* if, each READ(x) returns the value written by the most recent preceding WRITE(x, \cdot) in the sequence (or the initial value if no such WRITE exists). Other shared objects will be defined later as needed. Any sequence of operations on a collection of objects is *valid* if for each object, the subsequence of operations on that object is valid.

Partition Consistency is a class of abstract memory consistency models that is designed to capture processes that communicate by reading and writing shared variables. These variables are partitioned into subsets. Strong consistency is required within each subset, but not between different subsets. More formally, let $K = \{V_1, \dots, V_m\}$ be a partition of a subset of the set V of shared variables.

Definition 2.1. $\text{PC}(K)[C] \stackrel{\text{def}}{=} \exists \langle \text{valid total order } (O|_p \cup O|_{\text{writes}}, \xrightarrow{L_p}) : p \in P \rangle \text{ satisfying}$
 $(\forall p \in P : \text{Extends}[O|_p \cup O|_{\text{writes}}, \xrightarrow{L_p}, \xrightarrow{\text{prog}}]) \text{ and } (\forall p, q \in P, i \in [1, m] : \text{Agree}[O|_{\text{writes}}(V_i), \xrightarrow{L_p}, \xrightarrow{L_q}])$

Different instantiations of K yield different memory consistency models including several well-known models. For example, SC is $\text{PC}(\{V\})$, Pipelined-RAM is $\text{PC}(\emptyset)$, and PC-G is $\text{PC}(G)$ where $G = \{\{v\} \mid v \in V\}$.

¹Since p could be multithreaded, $(O|_p, \xrightarrow{\text{prog}})$ is not necessarily a total order.

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 is denoted $V|_{\text{multi-writers}}$. If $\{x\} \in K$ and x is a single-writer variable, then the Agree property for the set $\{x\}$ holds automatically because write operations on x are totally ordered by program order, which is itself preserved by the Extends property. Thus $\{x\}$ can be removed from K while maintaining $\text{PC}(K)$. Because implementations spend resources to maintain the consistency of each set in K , removing $\{x\}$ from K could reduce partition maintenance overhead in an implementation. This motivates two new natural instantiations of Partition Consistency, $\text{WeakPC-G} \stackrel{\text{def}}{=} \text{PC}(G)$ where for each $v \in V|_{\text{multi-writers}}$, $\{v\} \in G$, and $\text{WeakSC} \stackrel{\text{def}}{=} \text{PC}(\{V|_{\text{multi-writers}}\})$. By the previous observation, WeakPC-G is equivalent to PC-G ; however WeakSC is strictly weaker than SC though still stronger than PC-G .

The Network Model captures a concrete reliable, message-passing asynchronous network of multi-threaded processes. Each process has a set of locally shared variables that threads within that process use to communicate with each other. The local accesses to locally shared variables are sequentially consistent². Threads of distinct processes communicate by sending and receiving messages, where messages from a sender to a receiver are received in the order sent.

This intuition of a network is formalized as follows. The shared objects are variables (shared between threads of the same process) and messages (shared between different processes). Messages have distinct identifiers, and support the send $\text{SEND}(s, d, m)$ (source, destination, message) and the receive $\frac{\text{RECV}()}{s,d,m}$ operations. A sequence of message operations is *valid* if it contains at most one SEND and at most one RECV of any message. Define the following relations on the set O of operations of a computation C :

- 1) Message causality: $x \xrightarrow{\text{MessageOrder}_C} y \stackrel{\text{def}}{=} x, y \in O \wedge x = \text{SEND}(s, d, m) \wedge y = \frac{\text{RECV}()}{s,d,m}$; and
- 2) FIFO channel causality: $x \xrightarrow{\text{FifoChannel}_C} y \stackrel{\text{def}}{=} x, y \in O \wedge x = \frac{\text{RECV}()}{s,d,m} \wedge y = \frac{\text{RECV}()}{s,d,m'} \wedge \text{SEND}(s, d, m) \xrightarrow{\text{prog}} \text{SEND}(s, d, m')$; and
- 3) Writes-into causality for variables: $x \xrightarrow{\text{WritesInto}_C} y \stackrel{\text{def}}{=} x, y \in O \wedge x = \text{WRITE}(w, z) \wedge y = \frac{\text{READ}(w)}{z}$; and
- 4) Happens-before: $\xrightarrow{\text{HappensBefore}_C} \stackrel{\text{def}}{=} (\xrightarrow{\text{prog}} \cup \xrightarrow{\text{MessageOrder}} \cup \xrightarrow{\text{FifoChannel}} \cup \xrightarrow{\text{WritesInto}})^+$.

The definition $\xrightarrow{\text{HappensBefore}_C}$ is based on Lamport's happens-before [14], but that definition considers sequential processes that communicate only by message passing. This definition adds shared memory communication between threads and is designed to incorporate weak consistency.

Definition 2.2. $\text{NW}[C] \stackrel{\text{def}}{=} \exists \langle \text{valid total order } (O|_p, \xrightarrow{L_p}) : p \in P \rangle : \text{satisfying}$

$$\forall p \in P : \text{Extends}[O|_p, \xrightarrow{L_p}, \xrightarrow{\text{HappensBefore}_C}] \text{ and } \frac{\text{RECV}()}{s,d,m} \in O \text{ if and only if } \text{SEND}(s, d, m) \in O.$$

This definition captures what we would expect of a reliable message-passing network that ensures FIFO channels between each pair of processors. It requires that each process's view of its own operations is consistent with $\xrightarrow{\text{HappensBefore}_C}$ order. Thus, if two operations by threads of process p are causally ordered, and even if the intermediate operations that cause that ordering are not visible to p , there must be a valid view of all p 's operations that does not conflict with that causal ordering. The last conjunct ensures that the received messages are exactly those that are sent.

Any instance of Partition Consistency could be constructed directly on the Network model. We obtain cleaner proofs and better abstraction, however, by introducing an intermediate level to separate the fact that processes broadcast write updates and apply them locally from how the broadcasting is managed.

The Partial Order Broadcast model is designed to capture a collection of multithreaded processes, where threads within each process communicate through shared variables, and updates are communicated

²Weakening this assumption of "local sequential consistency" is possible, but would require adding some additional thread synchronization.

can't Lamports have weak consistency?

between distinct processes using a one-to-all `BCAST` and a corresponding `DELIVER`. Every process delivers updates in an order that extends the program order of the corresponding broadcast. Furthermore, updates can be labelled. Processes agree on the delivery order of all updates with the same label; such agreement is not required for differently labelled updates.

The formal definition of this model uses shared variables (shared between threads of the same process) and shared update objects (shared between different processes). Each update object is unique and supports the operation `BCAST(u, l)` (broadcast update u with label l to all) and the operation $\frac{\text{DELIVER}()}{u, l}$ (deliver the update u). (The label, l , is a null value, \perp , for unlabelled updates.) A sequence of `BCAST` and `DELIVER` operations is *valid* if 1) no `DELIVER` precedes its corresponding `BCAST` (This restriction does not require the corresponding `BCAST` to be in the valid order), and, 2) no specific `DELIVER` occurs more than once. Define the deliver relation (a partial order) on the set O_C of operations of a computation C of the Partial Order Broadcast model by:

$$x \xrightarrow{\text{delorder}_C} y \stackrel{\text{def}}{=} x, y \in O_C \wedge x = \frac{\text{DELIVER}()}{u_1, l_1} \wedge y = \frac{\text{DELIVER}()}{u_2, l_2} \wedge \text{BCAST}(u_1, l_1) \xrightarrow{\text{prog}} \text{BCAST}(u_2, l_2).$$

Let $O|_{\text{delivers}(l)}$ denote the set of all deliver operations returning an update with label l .

Definition 2.3. $\text{POB}(L)[C] \stackrel{\text{def}}{=} \exists \langle \text{valid total order } (O|_p, \xrightarrow{L_p}) : p \in P \rangle :$
 $(\forall p \in P : \text{Extends}[O|_p, \xrightarrow{L_p}, \xrightarrow{\text{prog}} \cup \xrightarrow{\text{delorder}}])$ and $(\forall p, q \in P, l \in L : \text{Agree}[O|_{\text{delivers}(l)}, \xrightarrow{L_p}, \xrightarrow{L_q}])$ and $\text{BCAST}(m, l) \in O$ if and only if $\frac{\text{DELIVER}()}{m, l} \in O|_p$.

This definition captures what we described as the intermediate Partial Order Broadcast model. It requires that each process's view of its own operations is a sequence that extends the program order of its own threads and delivers message according to the program order of the broadcasts. It also requires agreement between process's views of delivers of updates with the same label. The last conjunct ensures that every process delivers exactly the updates that were broadcast.

3 Implementations

Our implementations transform code for a *specified* model to code for a *target* model. For clarity, `SMALL CAPS` font is used to denote specification level operations; `Teletype` is used in to denote target level operations. To emphasize that a component belongs to the target level its name is sometimes annotated with a “hat” as in $\widehat{\text{name}}$.

This section first presents the slow-write/fast-read pseudo-code for implementing Partition Consistency on Partial Order Broadcast, The similar fast-write/slow-read version is in the full version of the paper [5]. Next is the pseudo-code for both the token-ring and queue implementations of the Partial Order Broadcast model on the Network model. The composition yields an implementation of any model in the abstract Partition Consistency class, including PC-G, Pipelined RAM, WeakSC, or SC, where the partition of the shared variables specifies the required agreement, on a reliable message passing network with FIFO channels.

Implementing Partition Consistency on Partial Order Broadcast

Our implementation of Partition Consistency using the Partial Order Broadcast model is a generalization of the way that Totally Ordered Broadcast is used to implement SC. $\text{POB}(L)$ enforces a similar agreement on the deliveries of updates with the same label as $\text{PC}(K)$ enforces on writes to variables within the same set in K .

Partition Consistency models processes that interact by reading and writing globally shared variables that have only weak consistency guarantees. Our task is to transform each such specified process p to a target process \widehat{p} for the Partial Order Broadcast model, where inter-process communication is via the partial order broadcast primitive. We achieve this by 1) creating a label for each partition in K , that is

added labels explicitly
 here — can this work
 later?

$L(K) = \{i : V_i \in K\}$; 2) mapping each p to a thread $\widehat{p}.m$ in the $\text{POB}(L(K))$ model; and 3) adding to each \widehat{p} a delivery thread $\widehat{p}.d$.

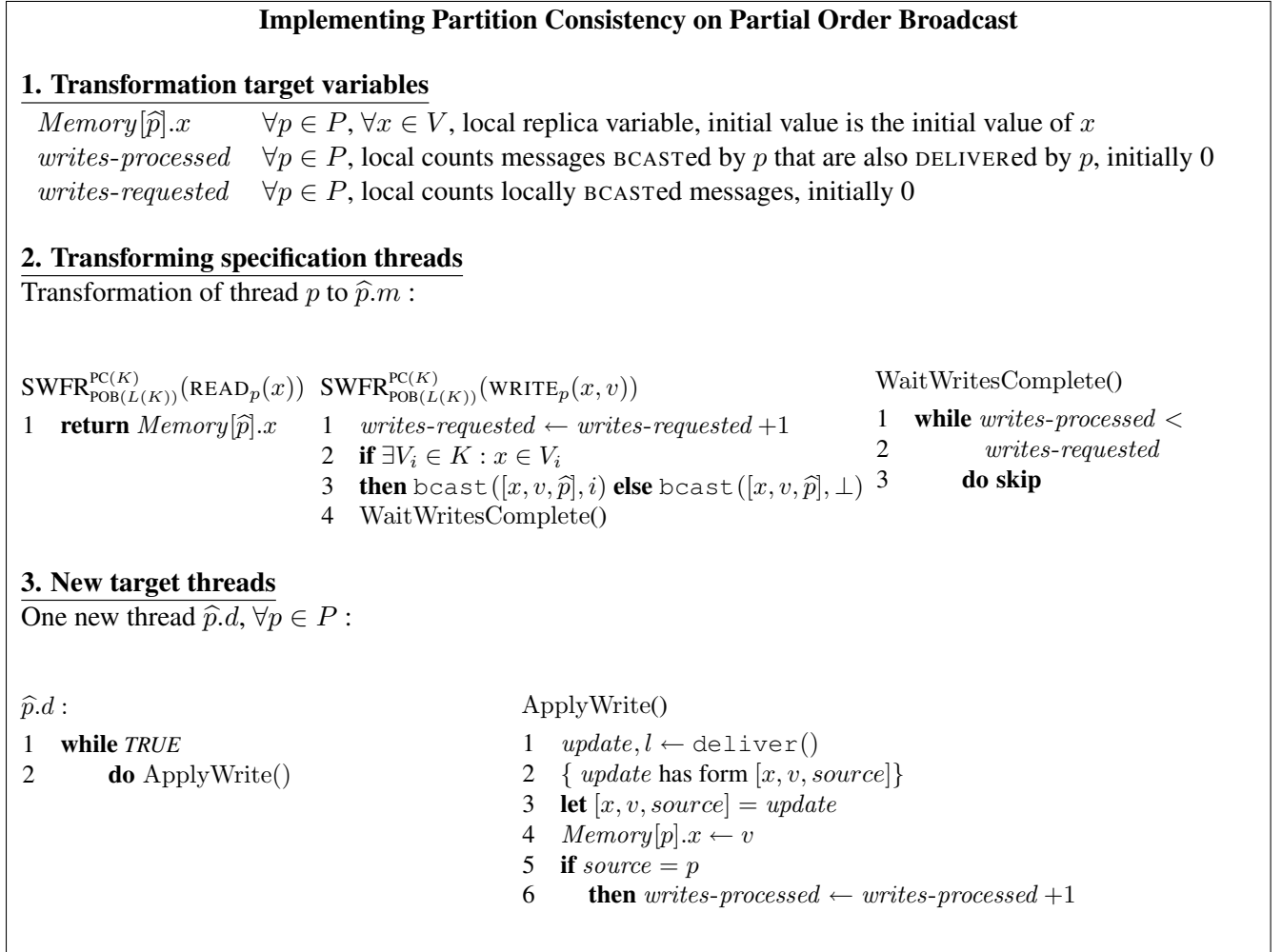


Figure 1: Implementing Partition Consistency on Partial Order Broadcast model

The main thread, $\widehat{p}.m$, is identical to p except that each `READ` and `WRITE` to a shared variable is replaced by a subroutine call. The transformation of a `READ` simply returns the value stored in \widehat{p} 's local memory. The transformation of a `WRITE` creates a `bcast` operation to be `delivered` to each target process. It has a label corresponding to the partition of the variable being written, if this partition exists.

The delivery thread, $\widehat{p}.d$, manages the `deliver` operations and maintains synchronization with $\widehat{p}.m$ via locally shared variables. It indefinitely applies updates to the local memory it shares with $\widehat{p}.m$. The procedure `WaitWritesComplete` causes $\widehat{p}.m$ to wait until $\widehat{p}.d$ has applied all the `WRITES` previously broadcast by $\widehat{p}.m$.

Under the $\text{SWFR}_{\text{POB}(L(K))}^{\text{PC}(K)}$ transformation, each process has at most one outstanding local write, since every write must be applied locally before the subroutine completes. Every write contains a wait, making these writes “slow”. An alternative is to move the `WaitWritesComplete` call from the end of the `WRITE` to the beginning of the `READ`. This gives us a fast-write/slow-read ($\text{FWSR}_{\text{POB}(L(K))}^{\text{PC}(K)}$) implementation described in the full version [5].

Implementating Partial Order Broadcast on Network using tokens $\text{TKN}_{\text{NW}}^{\text{POB}(L)}$

The Partial Order Broadcast model is very similar to the Network model: we need only specify how to implement `BCAST` and `DELIVER` by `send`ing and `recv`ing messages. The operations `READ` and `WRITE` are simply mapped through with an identity transformation. The processes in P are numbered starting at 0, and organized into a virtual ring such that $\text{next}(p) = (p + 1) \bmod |P|$. A token is created for each label $l \in L$, and for each token, a thread is created on each process to manage it.

To broadcast an update with a \perp label, we `send` it to every other process since these updates only require FIFO ordering. Updates with a non- \perp label need to maintain a per-label total order. For such updates, we acquire the label's token, then `send` the update to every other process, and wait for acknowledgements before continuing.

Each call to `PassToken` manages one acquisition and subsequent release of a token. It calls the `Guard` subroutine each time it acquires a token. The `Guard` returns only after handshaking with `ProtectedBcast` to determine that the token is no longer needed and can be released. $(pattern) \dashv \text{recv}()$ is pseudo-code that blocks until a message matching *pattern* is received and stored in the appropriate pattern variables.

The transformation of `BCAST` acquires the appropriate token to broadcast a labeled message by synchronizing with its token thread. The broadcast of a labeled message further requires that all processes deliver the message before the token is released. To ensure this, the implementation of `BCAST` waits for acknowledgments from all the processes before completing. To avoid deadlock this transformation requires that `BCASTS` and `DELIVERS` are invoked by separate threads.

Unlabeled messages are broadcast without acquiring a token.

Implementating Partial Order Broadcast on Network using timestamps $\text{TS}_{\text{NW}}^{\text{POB}(L)}$

Instead of strongly ordering the deliveries in time using tokens, the $\text{TS}_{\text{NW}}^{\text{POB}}$ implementation uses timestamps to enforce an agreement on the order of deliveries. This implementation generalizes Attiya and Welch's timestamp total order broadcast implementation [4].

The implementation uses priority queues, but since each one is only accessed by one thread, they can be constructed out of variables. One queue is created for each message label and one for unlabeled messages. Priority queues ensure that when a message is dequeued, it has the least timestamp out of the remaining messages in the same queue. If we further ensure that all messages with a smaller or equal timestamp have been received and `pri-enqueue`'d before `extract`ing a message, then all processes will deliver messages with the same label in the same order. Because messages are spread across the multiple priority queues, FIFO delivery order is not automatically enforced. To satisfy FIFO delivery order, we require that a message is only delivered by a process if it has the next counter value after the counter of the last delivered message from that source.

This implementation requires that for each process, at most one thread performs `DELIVER`. The first piece is the transformation of `BCAST`. Since the actual broadcast requires modifying the priority queue, we pass this responsibility on to the transformation of the `DELIVER` operation to avoid race conditions and more complicated synchronization.

The `DELIVER` transformation does the heavy lifting in this implementation. It drives the processing of messages by calling `HandleMessage` when it cannot return anything. The `HandleMessage` function handles all the background maintenance work, sending timestamp updates, modifying the timestamp and counter arrays, and queueing messages.

Messages contain labels indicating which queue they belong to, as well as other fields needed for the protocol. Unlabeled messages are queued in the special queue *queue \perp* . The notation *m.ts* denotes the *timestamp* field of a `ORD-MSG` or `TS-UPDATE` message *m*. The notation *m.src* denotes the message source field of an `ORD-MSG`. The notation *m.counter* denotes the program counter field of an `ORD-MSG`.

Using Tokens to Implement Partial Order Broadcast on Network

1. Transformation target variables

$need-token_l$ for each $l \in L$ and $p \in P$, handshake variable
 $door_l$ for each $l \in L$ and $p \in P$, handshake variable
 \hat{x} for each $x \in V$, identity transform

2. Transforming specification threads

Transformation of thread $p.m$ to $\hat{p}.m$ and $p.d$ to $\hat{p}.d$:

$TKN_{NW}^{POB(L)}(BCAST(u, l))$

```

1  if  $l \neq \perp$ 
2    then ProtectedBcast( $need-token_l, door_l, u, l$ )
3    else bcastop( $u, l$ )
    
```

$TKN_{NW}^{POB(L)}(DELIVER())$

```

1  ( $q, p, [MESSAGE, u, l]$ )  $\dashv$  recv()
2  if  $l \neq \perp$ 
3    then send( $p, q, [ACK]$ )
4  return  $u, l$ 
    
```

$TKN_{NW}^{POB(L)}(READ(x))$

```

1  return  $\hat{x}$ 
    
```

$TKN_{NW}^{POB(L)}(WRITE(x, v))$

```

1   $\hat{x} \leftarrow v$ 
    
```

ProtectedBcast($need, doorOpen, m, l$)

```

1   $need \leftarrow TRUE$ 
2  while  $\neg doorOpen$  skip
3  bcastop( $u, l$ )
4   $doorOpen \leftarrow FALSE$ 
5   $need \leftarrow FALSE$ 
    
```

bcastop(u, l)

```

1  forall  $q \in P$ 
2    do send( $p, q, [MESSAGE, u, l]$ )
3  if  $l \neq \perp$ 
4    {Wait for acknowledgment}
5    then forall  $q \in P$ 
6      do ( $q, p, [ACK]$ )  $\dashv$  recv()
    
```

3. New target threads

One new token thread $\hat{p}.TokenThread_l, \forall p \in P, l \in L$:

$\hat{p}.TokenThread_l$:

```

1  if  $\hat{p} = 0$ 
2    then send( $\hat{p}, next(\hat{p}), [TOKEN, BCASTGROUPTOKEN_l]$ )
3  loop
4    do PassToken $_{\hat{p}}(l)$ 
    
```

next(\hat{p})

```

1  return  $(\hat{p} + 1) \bmod |\hat{P}|$ 
    
```

PassToken $_p(l)$

```

1  ( $q, p, [TOKEN, BCASTGROUPTOKEN_l]$ )  $\dashv$  recv()
2  Guard( $need-token_l, door_l$ )
3  send( $p, next(p), [TOKEN, BCASTGROUPTOKEN_l]$ )
    
```

Guard($need, doorOpen$)

```

1  if  $need$ 
2    then  $doorOpen \leftarrow TRUE$ 
3    while  $need$  skip
    
```

Figure 2: Token implementation of Partial Order Broadcast

Using Timestamps to Implement Partial Order Broadcast on Network

1. Transformation target variables

counter for each $p \in P$, local array of last delivered counter value for each process, initially all 0
T for each $p \in P$, local array of last received timestamp value for each process, initially all 0
queue_l for each $l \in L$ and $p \in P$ local priority queue for messages labeled l , initially empty
fifo for each $p \in P$, local array of queues for unlabeled messages, one queue for each process, initially all empty
 \hat{x} for each $x \in V$, \hat{x} identity transform of x

2. Transforming specification threads

Transformation of thread $p.m$ to $\hat{p}.m$ and $p.d$ to $\hat{p}.d$:

$\text{TS}_{\text{NW}}^{\text{POB}(L)}(\text{READ}(x))$ 1 return \hat{x}	$\text{TS}_{\text{NW}}^{\text{POB}(L)}(\text{WRITE}(x, v))$ 1 $\hat{x} \leftarrow v$	$\text{TS}_{\text{NW}}^{\text{POB}(L)}(\text{BCAST}(\text{update}, l))$ 1 $\text{send}(\hat{p}, \hat{p}, [\text{LOCAL-BROADCAST-REQUEST}, \text{update}, l])$
--	---	--

$\text{TS}_{\text{NW}}^{\text{POB}(L)}(\text{DELIVER}())$ { l can be \perp } 1 while $(\neg \exists l \in L : \text{CanExtract}(\text{queue}_l))$ 2 $\wedge (\neg \exists \hat{p} \in \hat{P} : \text{CanDequeue}(\text{fifo}[\hat{p}]))$ 3 do $\text{HandleMessage}()$ 4 case (choose $(Ll : \text{CanExtract}(\text{queue}_l))$ 5 $ (U\hat{p} : \text{CanDequeue}(\text{fifo}[\hat{p}]))$ 6 of 7 (Ll) then $m \leftarrow \text{extractmin}(\text{queue}_l)$ 8 $(U\hat{p})$ then $m \leftarrow \text{dequeue}(\text{fifo}[\hat{p}])$ 9 $\text{counter}[m.\text{src}] \leftarrow m.\text{counter}$ 10 return $m.\text{update}, m.l$	$\text{HandleMessage}_p()$ 1 $\hat{s}, \hat{p}, \text{message} \leftarrow \text{recv}()$ 2 case message of: 3 $[\text{LOCAL-BROADCAST-REQUEST}, \text{update}, l]$ 4 then $T[\hat{p}] \leftarrow T[\hat{p}] + 1$ 5 $\text{local-counter} \leftarrow \text{local-counter} + 1$ 6 $\text{premessage} \leftarrow [{}_{T[\hat{p}], \text{local-counter}, \hat{p}}^{\text{ORD-MSG}, \text{update}, l}]$ 7 $\text{ProcessMessage}(\text{premessage}, l)$ 8 $\text{FifoBroadcast}(\text{premessage})$ 9 $[\text{TS-UPDATE}, \text{timestamp}, \hat{q}]$ 10 then $T[\hat{q}] \leftarrow \text{timestamp}$ 11 $[\text{ORD-MSG}, \text{update}, l, \text{timestamp}, \text{counter}, \hat{q}]$ 12 then $T[\hat{q}] \leftarrow \text{timestamp}$ 13 $\text{ProcessMessage}(\text{message}, l)$ 14 if $\text{timestamp} > T[\hat{p}]$ 15 then $T[\hat{p}] \leftarrow \text{timestamp}$ 16 $\text{FifoBroadcast}([\text{TS-UPDATE}, T[\hat{p}], \hat{p}])$
---	---

3. New target threads

No new target threads for this implementation.

Figure 3: Timestamp implementation of Partial Order Broadcast

The implementation of FifoBroadcast relies on the underlying model to provide FIFO properties. Reads and writes are mapped through with an identity transform. \hat{x} is used to emphasize that the model has changed.

The parameter L of the transform $\text{TS}_{\text{NW}}^{\text{POB}(L)}$ is needed since a priority queue is created for each label in L . To clarify the presentation, we will often drop the parameter and just write $\text{TS}_{\text{NW}}^{\text{POB}}$, as L will be provided by the specification $\text{POB}(L)$.

CanExtract(<i>queue</i>)	
1 if pri-isempty(<i>queue</i>)	ProcessMessage(<i>message</i> , <i>l</i>)
2 then return FALSE	1 if $l \neq \perp$
3 else $m \leftarrow \text{peekmin}(\text{queue})$	2 then pri-enqueue(<i>queue</i> _{<i>l</i>} , <i>message</i>)
4 return ($m.\text{counter} = \text{counter}[m.\text{src}] + 1$)	3 else enqueue(<i>fifo</i> [<i>message</i> . <i>src</i>], <i>message</i>)
$\wedge (\forall \hat{q} \in \hat{P} : m.\text{ts} < T[\hat{q}])$	
CanDequeue(<i>fifo-queue</i>)	
1 if isempty(<i>fifo-queue</i>)	FifoBroadcast(<i>message</i>)
2 then return FALSE	1 for $\hat{q} \in \hat{P} \setminus \{\hat{p}\}$
3 else $m \leftarrow \text{peek-head}(\text{fifo-queue})$	2 do send($\hat{p}, \hat{q}, \text{message}$)
4 return ($m.\text{counter} = \text{counter}[m.\text{src}] + 1$)	

Figure 4: Auxiliary functions for Timestamp implementation

4 Proofs of Correctness of the Implementations

4.1 Proof structure and setup

Section 3 presented transformations of a specified multiprogram into a target multiprogram. To prove correctness, we must show the possible computations of these two multiprograms that can arise from their respective memory consistency model, have the same “outcome”. We make this precise as follows. Let $\tau(P)$ denote a transformation of multiprogram P . The possible computations of multiprogram P (respectively, $\tau(P)$) on the specified (respectively, target) memory consistent model MC (respectively, $\widehat{\text{MC}}$) is the set $\mathcal{C}(P, \text{MC})$ (respectively, $\mathcal{C}(\tau(P), \widehat{\text{MC}})$). But $\tau(P)$ transforms specified operation invocations that require a response into subroutines that return a response. So these returned responses can be used to *interpret* each computation in $\mathcal{C}(\tau(P), \widehat{\text{MC}})$ as a computation of P . We need to show that each such interpreted computation could have arisen in the specified model. That is, we must show that the interpretation of any computation in $\mathcal{C}(\tau(P), \widehat{\text{MC}})$ is in $\mathcal{C}(P, \text{MC})$. If this is satisfied for any P , we say that $\tau(P)$ *correctly implements* MC on $\widehat{\text{MC}}$. Figure 16 (at the end of this paper) depicts this proof obligation for the transformations in Section 3.

We show that a computation satisfies a model by constructing a collection of orders that jointly satisfy the model’s constraints. We call collections of orders showing that a computation satisfies a memory consistency model, *witness orders* to that consistency model.

The proofs, including all four in the full paper, have the following structure:

Assume: $\hat{C} \in \mathcal{C}(\tau(P), \widehat{\text{MC}})$. Let $C \in \mathcal{C}(P)$ be the interpretation of \hat{C} .

Build: Choose some collection of *witness orders* \hat{A} that satisfy $\widehat{\text{MC}}[\hat{C}]$. Use \hat{A} to construct a corresponding collection of orders A for C .

Verify: Show that the collection A are witness orders to $\text{MC}[C]$.

In this paper, we consider only *finite* computations, and completed computations produced by transformations of these computations. For long-lived computations, we would need to allow only a prefix of the required messages or updates to be RECV’d or DELIVER’d, order operations not seen by certain processes in their views and handle these changes in the proofs.

We prove the required lemmas assuming the computations, orders and constructed orders of the **Assume**

and **Build** steps of the theorem. In developing these proofs, we found that a diagram format helped uncover many underlying assumptions and errors in the proofs. This diagram format also allowed us to quickly work out proofs on a whiteboard. Before presenting the lemmas we digress to describe the proof diagram format.

4.2 Proof diagrams

Edges in the diagram represent boolean expressions and the diagram is interpreted as the conjunction of these expressions. The basic building blocks are:

$$\begin{array}{l}
 a \text{ --- } b \text{ asserts that } a = b. \\
 a \xrightarrow{L} b \text{ asserts that } a \xrightarrow{L} b. \\
 a \text{ --- } \overset{R}{\curvearrowright} b \text{ asserts that } a \text{ is related to } b \text{ by a relation } R.
 \end{array}
 \qquad
 \begin{array}{c}
 a \xrightarrow{L} b \text{ asserts that } (a \xrightarrow{L} b) \implies (c \xrightarrow{M} d). \\
 \downarrow \\
 c \xrightarrow{M} d
 \end{array}$$

For sets A and B , using notation similar to Lamport's system executions [16] we have:

$$\begin{array}{l}
 A \xrightarrow{L} B \text{ asserts that } \forall a \in A, b \in B : a \xrightarrow{L} b. \quad A \vdash \overset{L}{\succ} B \text{ asserts that } \exists b \in B : \forall a \in A : a \xrightarrow{L} b. \\
 A \text{ --- } \overset{L}{\succ} B \text{ asserts that } \exists a \in A : \forall b \in B : a \xrightarrow{L} b. \quad A \text{ --- } \prec B \text{ asserts that } A \subseteq B.
 \end{array}$$

Multiple edges between two nodes denote the disjunction of the corresponding assertions. So $a \overset{E}{\curvearrowright} b$ asserts that $a \xrightarrow{D} b$ or $a \xrightarrow{E} b$. The direction of the edge will indicate which way it should be read. For example $a \xrightarrow{L} b$ and $b \xleftarrow{L} a$ are equivalent.

4.3 Correctness of the $\text{SWFR}_{\text{POB}(L(K))}^{\text{PC}(K)}$ and $\text{FWSR}_{\text{POB}(L(K))}^{\text{PC}(K)}$ implementations

The proofs of $\text{FWSR}_{\text{POB}}^{\text{PC}}$ and $\text{SWFR}_{\text{POB}}^{\text{PC}}$ are very similar, they only differ in one step. This step can be treated generically, so we present one proof for both implementations. Let $\text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}$ refer to either implementation.

Let P be a program compatible with the $\text{PC}(K)$. This means that P must be a single threaded program that only operates on read/write variables.

Theorem 4.1. $\text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}$ correctly implements $\text{PC}(K)$ on $\text{POB}(L(K))$ for any K .

Begin Proof

Assume: Let \widehat{C} be a computation in $\mathcal{C}(\text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(P), \text{POB}(L(K)))$ and let C be a computation in $\mathcal{C}(P)$ such that \widehat{C} is an interpretation of C . Let \widehat{O} denote the set of operations $O_{\widehat{C}}$.

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

Let $\langle (O_{\widehat{C}} | \widehat{p}, \xrightarrow{\widehat{L}_p}) : p \in \text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(P) : \rangle$ be witness orders to $\text{POB}(L(K))[\widehat{C}]$.

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

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

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

letting:

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

- (b) Every $\text{write}(\text{Memory}[p].x, v)$ must have a $o_d = \frac{\text{DELIVER}()}{[\text{WRITE}, x, v]}$ in the same $\text{ApplyWrite}()$ call. This deliver operation o_d must have a corresponding $\text{BCAST}(m)$, which can only have occurred in the transformation of some high level write $\text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(\text{WRITE}(x, v))$. Associate the original operation with this specification level one by letting:

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

2. Build the sequence $\text{Short}(\widehat{L}_{\widehat{p}})$ 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 $\text{Short}(\widehat{L}_{\widehat{p}})$ is a local memory read or write on a replica so we replace them with their associated high level operations. This sequence induces the required total order.

Verify: We prove that the constructed orders $\xrightarrow{L_p}$ are witnesses to $\text{PC}(K)[C]$.

To show this, we directly prove each of the properties in Definition 2.1.

The constructed sequences $\langle L_p : p \in P : \rangle$ induce corresponding total orders

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

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

Constraint	Lemma
$\text{Extends}[\xrightarrow{L_p}, \xrightarrow{\text{prog}}, O p \cup O \text{writes}]$	Lemma 4.2
$\forall p, q \in P, i \in [1, k] : \text{Agree}[\xrightarrow{L_p}, \xrightarrow{L_q}, O \text{writes}(S_i)]$	Lemma 4.3

Therefore $\text{PC}(K)[C]$ as required.

End Proof

Let $\widehat{C} \in \mathcal{C}(\text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(P), \text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(J), \text{POB}(L(K)))$. Let C be such that \widehat{C} is an interpretation of C . This matches the **Assume** step of Theorem 4.1.

Let $\langle (O_{\widehat{C}}|\widehat{p}, \xrightarrow{\widehat{L}_{\widehat{p}}}) : p \in \text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(P) : \rangle$ be witness orders that satisfy $\text{POB}(L(K))[\widehat{C}]$. Construct $\langle \xrightarrow{L_p} : p \in P : \rangle$ based on the chosen witness orders as in the **Build** step of Theorem 4.1.

Lemma 4.2. $\forall p \in P : \text{Extends}[\xrightarrow{L_p}, \xrightarrow{\text{prog}}, O|p \cup O|\text{writes}]$

Begin Proof

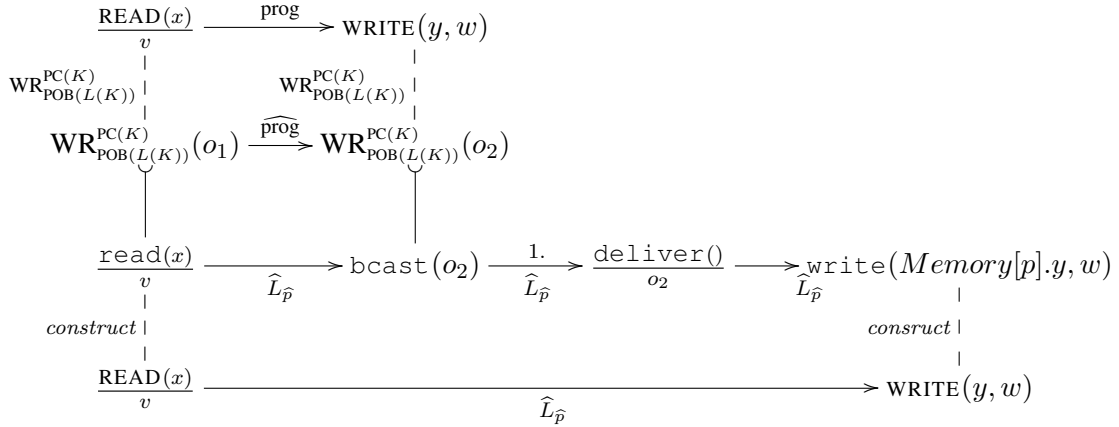
Let o_1, o_2 be operations in $O|p \cup O|\text{writes}$ so that $o_1 \xrightarrow{\text{prog}} o_2$. Since they are ordered by program order, they must have been invoked by the same process, so $\text{proc}(o_1) = \text{proc}(o_2) = q$ for some q . $\text{PC}(K)$ only has read/write variable objects, so there are four cases for o_1, o_2 :

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

We have $p = q$ and $\text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(o_1) \xrightarrow{\text{prog}} \text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(o_2) \implies o_1 \xrightarrow{L_p} o_2$ since the reads are directly translated, they remain in program order.

Case 2: read, write Then $o_1 = \frac{\text{READ}(x)}{v}$ and $o_2 = \text{WRITE}(y, w)$:

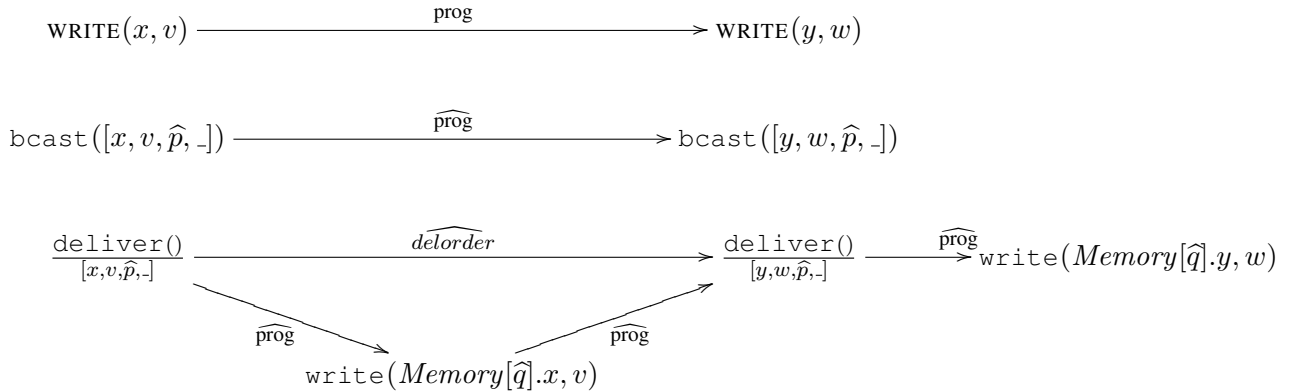
We have $p = q$ and $\text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(o_1) \xrightarrow{\widehat{\text{prog}}} \text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(o_2)$ which implies



1. $\text{POB}(L(K))$ that delivers must follow their corresponding broadcasts in the local order.

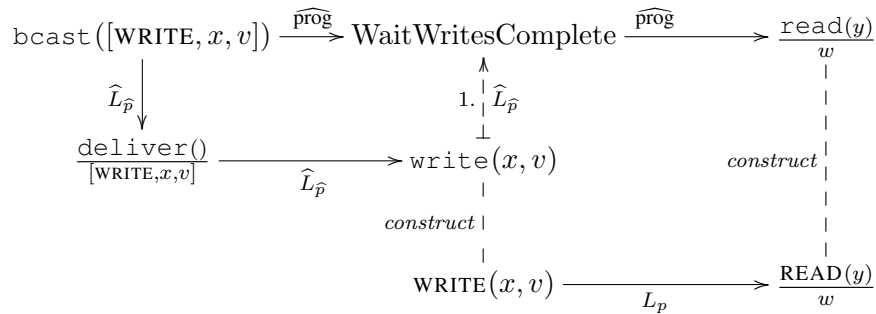
Case 3: write, write Then $o_1 = \text{WRITE}(x, v)$ and $o_2 = \text{WRITE}(y, w)$

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 $o_1 = \text{WRITE}(x, v)$ and $o_2 = \frac{\text{READ}(y)}{w}$

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 L_p extends $\widehat{\text{prog}}$.

End Proof

Lemma 4.3. $\forall p, q \in P, i \in [1, k] : \text{Agree}[\xrightarrow{L_p}, \xrightarrow{L_q}, O|\text{writes}(S_i)]$

— Begin Proof —

Let W_1, W_2 be write operations in some $O|\text{writes}(S_i)$. Then the broadcasts of these writes in the transformation are $\text{bcast}(msg_{w_1}) \in \text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(W_1)$ and $\text{bcast}(msg_{w_2}) \in \text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(W_2)$. 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 $w_{1p} \xrightarrow{\widehat{L}_{\widehat{p}}} w_{2p}$, then $\frac{\text{deliver}()}{msg_{w_1}} \xrightarrow{\widehat{L}_{\widehat{p}}} w_{1p} \xrightarrow{\widehat{L}_{\widehat{p}}} \frac{\text{deliver}()}{msg_{w_2}} \xrightarrow{\widehat{L}_{\widehat{p}}} w_{2p}$ which implies $\frac{\text{deliver}()}{msg_{w_1}} \xrightarrow{\widehat{L}_{\widehat{q}}} w_{1q} \xrightarrow{\widehat{L}_{\widehat{q}}} \frac{\text{deliver}()}{msg_{w_2}} \xrightarrow{\widehat{L}_{\widehat{q}}} w_{2q}$ since $\text{POB}(L(K))$ requires that deliveries of messages with the same label must agree.

— End Proof —

4.4 Correctness of $\text{TKN}_{\text{NW}}^{\text{POB}(L)}$ transformation

Let P be a multiprogram that uses READS, WRITES, BCASTS, and for each process, only calls DELIVER from at most one thread, calls BCAST from at most one thread, and BCAST and DELIVER are never called by the same thread.

Theorem 4.4. $\text{TKN}_{\text{NW}}^{\text{POB}(L)}$ correctly implements $\text{POB}(L)$ on NW, for any L .

— Begin Proof —

Assume:

Let \widehat{C} be a computation in \mathcal{C} ($\text{TKN}_{\text{NW}}^{\text{POB}(L)}(P)$, NW) and let C be an interpretation of \widehat{C} .

To show $\text{POB}(L)[C]$ we construct witness orders that satisfy the requirements of Definition 2.3.

Build:

Choose some collection of witness orders $\langle (O|\widehat{p}, \xrightarrow{\widehat{L}_{\widehat{p}}}) : \widehat{p} \in \text{TKN}_{\text{NW}}^{\text{POB}(L)}(P) \rangle$ to $\text{NW}[\widehat{C}]$.

Recall that for a total order (O, \xrightarrow{X}) , its induced sequence is denoted 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 $(\widehat{O}|\widehat{p}, \xrightarrow{\widehat{L}_{\widehat{p}}})$ to $\text{NW}[\widehat{C}]$ and form $\text{Short}(\widehat{L}_{\widehat{p}})$ by removing:
 1. all operations on the handshake variables $need\text{-}token_l$ and $door_l$ for all $l \in L$.
 2. all send and recv operations except: the first send of a $\text{bcastop}()$ and the recv of a DELIVER.

Note that each remaining operation was produced from the transformation of some specification level operation, rather than a thread created by the transformation.

- Associate the target level operations in $\text{Short}(\widehat{L}_{\widehat{p}})$ with specification level ones by defining $\sim_{\text{construct}}$ as follows:

For this proof we just have to associate these target level operations with the specification level operations that produced them.

Target Operation in transformation	Let
$\frac{\text{read}(x)}{v} \in \text{TKN}_{\text{NW}}^{\text{POB}(L)}(\frac{\text{READ}(x)}{v})$	$\frac{\text{read}(x)}{v} \underset{\text{construct}}{\sim} \frac{\text{READ}(x)}{v}$.
$\text{write}(x, v) \in \text{TKN}_{\text{NW}}^{\text{POB}(L)}(\text{WRITE}(x, v))$	$\text{write}(x, v) \underset{\text{construct}}{\sim} \text{WRITE}(x, v)$.
$\text{TKN}_{\text{NW}}^{\text{POB}(L)}(\text{BCAST}(m, l)) \cdot \text{send}(s, d, m)$	$\text{send}(s, d, m) \underset{\text{construct}}{\sim} \text{BCAST}(s, d, m)$.
$\frac{\text{recv}()}{s, d, m} \in \text{TKN}_{\text{NW}}^{\text{POB}(L)}(\frac{\text{DELIVER}()}{m, l})$	$\frac{\text{extractmin}(queue)}{m} \underset{\text{construct}}{\sim} \frac{\text{DELIVER}()}{m, l}$.

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

Verify:

The construction yields total orders $(O|p, \xrightarrow{L_p})$ that are valid by the fact that projecting on to subsets of objects preserves validity of the variables. The sequences of BCAST and DELIVER are valid since 1) the send of the BCAST $\xrightarrow{\text{HappensBefore}}$ the extractmin of its DELIVER and 2) each ORD-MSG is sent only once to each processor, received only once at each processor and enqueued only once at each processor.

The following table shows the properties we prove to verify the constraints of Definition 2.3:

Constraint	Lemma
$\forall p \in P : (O p, \xrightarrow{L_p})$ is a valid total order	By construction
$\forall p \in P : \text{Extends}[O p, \xrightarrow{L_p}, \xrightarrow{\text{prog}}]$	Lemma 4.5
$\forall p \in P : \text{Extends}[O p, \xrightarrow{L_p}, \xrightarrow{\text{delorder}}]$	Lemma 4.9
$\forall p, q \in P, l \in L : \text{Agree}[O \text{delivers}(l), \xrightarrow{L_p}, \xrightarrow{L_q}]$	Lemma 4.10

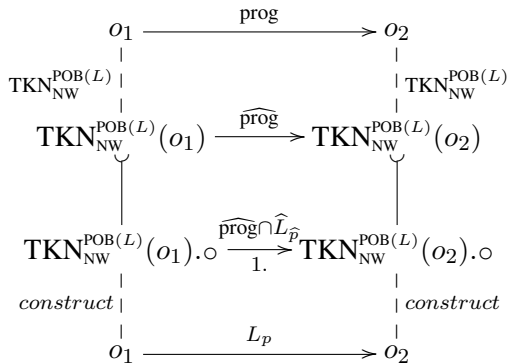
End Proof

Lemma 4.5. $\forall p \in P : \text{Extends}[O|p, \xrightarrow{L_p}, \xrightarrow{\text{prog}}]$

Begin Proof

This proof is simplified because of the way transformations are defined, the memory model, and the choice of operations in the construction of the $\xrightarrow{L_p}$ orders.

Let $o_1, o_2 \in O|p$ such that $o_1 \xrightarrow{\text{prog}} o_2$. Let $\text{TS}_{\text{NW}}^{\text{POB}(L)}(o) \cdot \circ$ be the operation in the transformation of o so that $\text{TS}_{\text{NW}}^{\text{POB}(L)}(o) \cdot \circ \underset{\text{construct}}{\sim} o$.



1. $\text{Extends}[O|\widehat{p}, \xrightarrow{\widehat{L}_{\widehat{p}}}, \xrightarrow{\widehat{\text{prog}}}]$

End Proof

Lemma 4.6. For all $l \in L$ let $T_l = \{\text{PassToken}_p() : \text{PassToken}_p() \subseteq O\}$. then $(B_l, \xrightarrow{\text{HappensBefore}})$ is a total order.

Begin Proof

Either

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

since there is only one original `send` for each token at the beginning of `TokenThreadl`, and each `PassToken.recv` has exactly one corresponding `send` which is either the original `send` or the `send` at the end of a `PassToken`.

End Proof

Lemma 4.7. *Let $p \in P$, $n, d \in V$ $T_p = \{\text{ProtectedBcast}(n, d, m, l) : \text{ProtectedBcast}(n, d, m, l) \subseteq O\}$ and $G_p = \{\text{Guard}(n, d) : \text{Guard}(n, d) \subseteq O\}$ Then there exists an injection `protected-by` : $T_p \mapsto G_p$ so that:*

`protected-by` (`ProtectedBcast`(n, d, m, l) = `Guard`(n, d)
if and only if

$$\text{ProtectedBcast}(n, d, m, l). \widehat{\text{bcastop}} \vdash - \triangleright \text{Guard}(n, d)$$

and

$$\text{ProtectedBcast}(n, d, m, l). \widehat{\text{bcastop}} \vDash - \triangleright \text{Guard}(n, d)$$

Begin Proof

Let

$$\text{protected-by}(\text{ProtectedBcast}(n, d, m, l)) \stackrel{\text{def}}{=} \text{Guard}(n, d) : \text{Guard}(n, d). \text{write}(d, \text{TRUE}) \xrightarrow{\widehat{\text{WritesInto}}} \text{ProtectedBcast}(n, d, m, l). \frac{\text{read}(d)}{\text{TRUE}}$$

Let $PrB = \text{ProtectedBcast}(n, d, m, l)$ and $G = \text{Guard}(n, d)$

$$G. \text{write}(d, \text{TRUE}) \xrightarrow{\widehat{\text{WritesInto}}} PrB. \frac{\text{read}(d)}{\text{TRUE}} \xrightarrow{\widehat{\text{prog}}} PrB. \text{bcastop}$$

$$G \text{ --- } \widehat{\text{HappensBefore}} \text{ --- } PrB. \text{bcastop}$$

Other:

$$\text{ProtectedBlock}^1. \text{bcastop} \xrightarrow{\widehat{\text{prog}}} \text{ProtectedBlock}^1. \text{write}(need, \widehat{\text{WritesInto}} \text{FALSE}) \xrightarrow{\widehat{\text{prog}}} G. \frac{\text{read}(need)}{\text{FALSE}}$$

$$\text{ProtectedBlock}^1. \text{bcastop} \vdash \text{---} \widehat{\text{HappensBefore}} \text{---} \triangleright G$$

We now show that $\text{ProtectedBlock}^1 = PrB$.

Since all the `ProtectedBlock` instances are called by the same thread they are totally ordered by program order. There are 3 cases:

Case $\text{ProtectedBlock}^1 = PrB$: done.

Case $PrB \xrightarrow{\widehat{\text{prog}}} \text{ProtectedBlock}^1$

$$PrB. \text{write}(need, \text{FALSE}) \xrightarrow[2]{\widehat{\text{prog}}} \text{ProtectedBlock}^1. \frac{\text{read}(need)}{\text{TRUE}} \xrightarrow{\widehat{\text{prog}}} \text{ProtectedBlock}^1. \text{WRITE}(need, \text{FALSE})$$

$$\begin{array}{c} \widehat{\text{prog}} \uparrow \\ PrB. \frac{\text{read}(doorOpen)}{\text{TRUE}} \\ L_p \uparrow 1. \end{array}$$

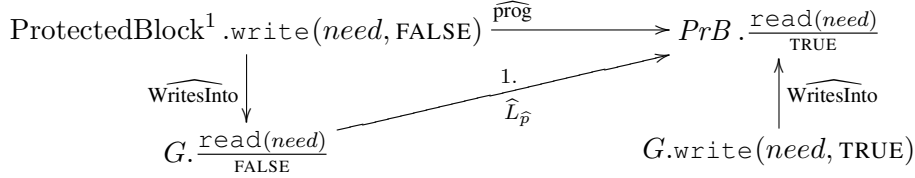
$$G. \text{write}(doorOpen, \text{TRUE})$$

$$\begin{array}{c} \downarrow 1. L_p \\ G. \frac{\text{read}(need)}{\text{FALSE}} \end{array}$$

1. G , PrB , and ProtectedBlock^1 were chosen to satisfy these properties.

2. Since $\xrightarrow{L_p}$ is valid, there must be a $\text{write}(need, \text{TRUE})$ between the $G.\text{write}(need, \text{TRUE})$ and $G.\frac{\text{read}(need)}{\text{FALSE}}$ but since all Guard calls on the same handshake variables are in the same thread, G must perform this write, which is not possible by the code. Contradiction.

Case $\text{ProtectedBlock}^1 \xrightarrow{\text{prog}} PrB$



1. $G.\text{write}(need, \text{TRUE})$ must be ordered between these two operations in $\widehat{L_{\hat{p}}}$, but then $G.\text{write}(need, \text{TRUE}) \xrightarrow{\text{prog}} G.\frac{\text{read}(need)}{\text{FALSE}}$ and $G.\frac{\text{read}(need)}{\text{FALSE}} \xrightarrow{\widehat{L_{\hat{p}}}} G.\text{write}(need, \text{TRUE})$. Contradiction.

protected-by is injective since all the Guard instances are in the same thread and are totally ordered by program order.

End Proof

Lemma 4.8. For all $l \in L$, let $B_l = \{TKN_{NW}^{\text{POB}(L)}(\text{BCAST}(u, l)).\text{bcastop} : \text{BCAST}(u, l) \in O\}$ then $(B_l, \xrightarrow{\widehat{\text{HappensBefore}}})$ is a total order.

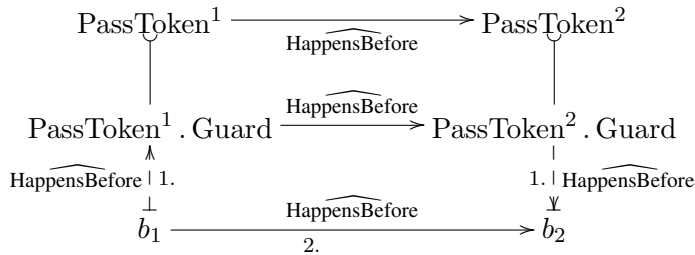
Begin Proof

Let $l \in L$, protected-by be a function provided by Lemma 4.7.

Let $b_1 = TKN_{NW}^{\text{POB}(L)}(\text{BCAST}(u_1, l)).\text{bcastop}$, $b_2 = TKN_{NW}^{\text{POB}(L)}(\text{BCAST}(u_2, l)).\text{bcastop}$ be distinct elements of B_l .

Then let $\text{PassToken}^1 = \text{protected-by}(b_1)$ and $\text{PassToken}^2 = \text{protected-by}(b_2)$.

By the Lemma 4.6, the PassToken instances are totally ordered. Suppose WLOG that $\text{PassToken}^1 \xrightarrow{\widehat{\text{HappensBefore}}} \text{PassToken}^2$ then:



1. By Lemma 4.7 and construction of PassToken^1 and PassToken^2 .

2. By 1 and the order on the Guard instances.

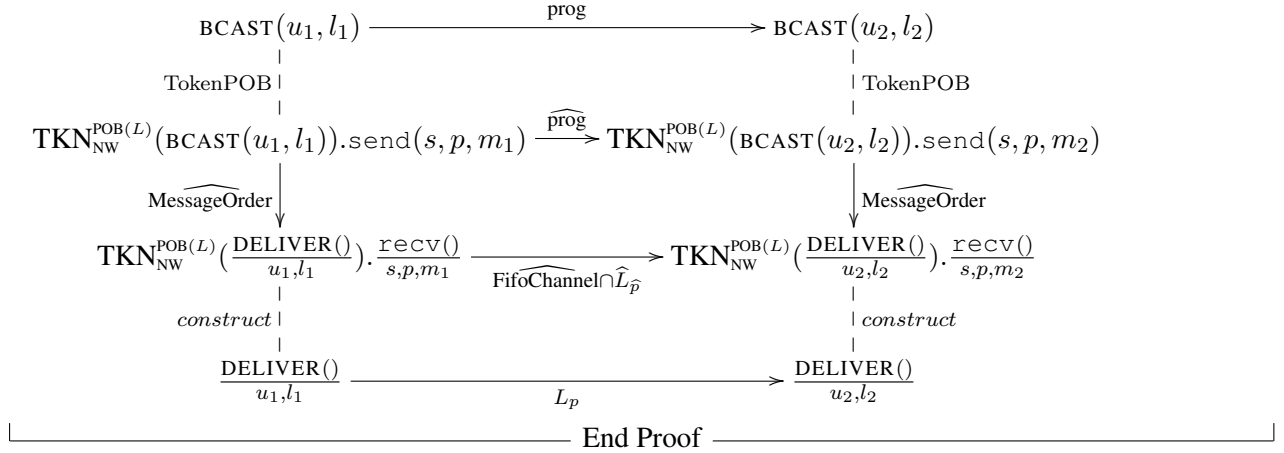
End Proof

Lemma 4.9. $\forall p \in P : \text{Extends}[O|p, \xrightarrow{L_p}, \xrightarrow{\text{delorder}}]$

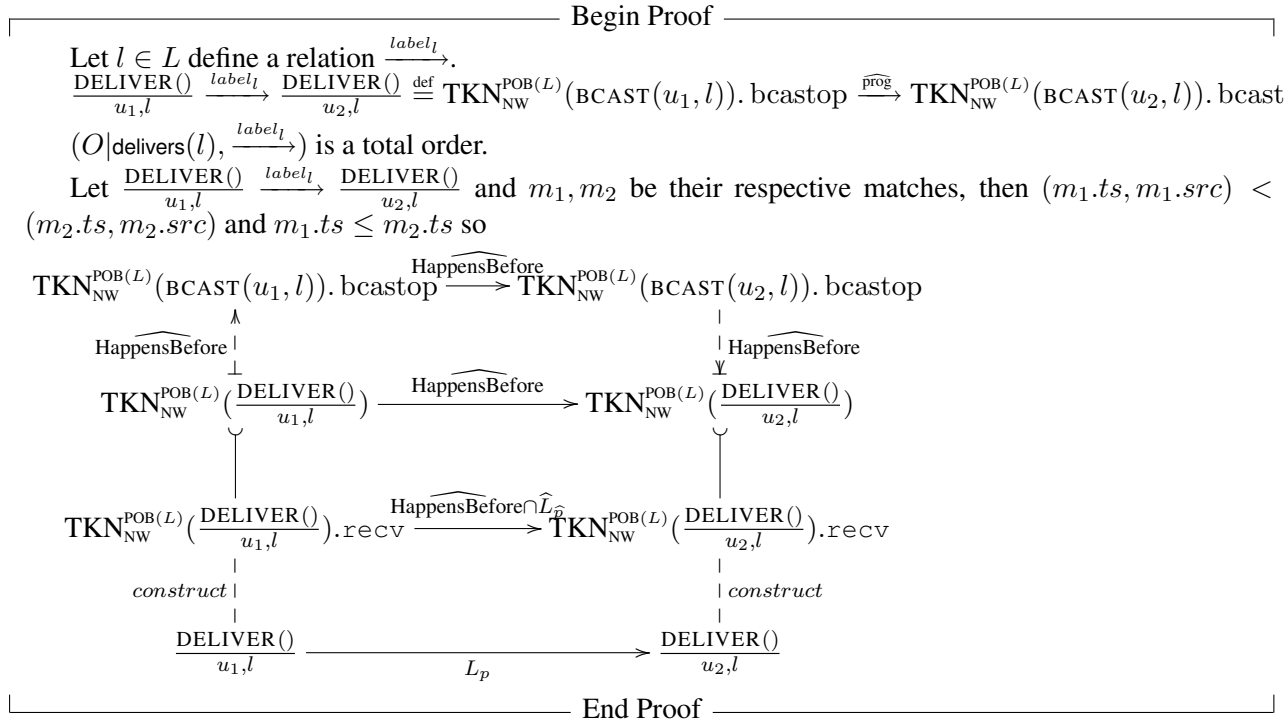
Begin Proof

Let $o_1, o_2 \in O|p$ such that $o_1 \xrightarrow{\text{delorder}} o_2$.

Then $o_1 = \frac{\text{DELIVER}()}{u_1, l_1}$, $o_2 = \frac{\text{DELIVER}()}{u_2, l_2}$ and $\text{BCAST}(u_1, l_2) \xrightarrow{\text{prog}} \text{BCAST}(u_2, l_2)$.



Lemma 4.10. $\forall p, q \in P, l \in L : \text{Agree}[O|\text{delivers}(l), \xrightarrow{L_p}, \xrightarrow{L_q}]$



4.5 Correctness of the Timestamp implementation

We illustrate these proof techniques by presenting (up to space constraints) the proof that the transformation $\text{TS}_{\text{NW}}^{\text{POB}(L)}(P)$ correctly implements a multiprogram P for the Partial Order Broadcast model, for any label set L , on the Network model. The proofs of correctness of the transformation from $\text{PC}(K)$ to $\text{POB}(L(K))$ for each of the transformations $\text{SWFR}_{\text{POB}}^{\text{PC}}$ and $\text{FWSR}_{\text{POB}}^{\text{PC}}$, generalize the ideas of Attiya and Welch [3] (corrected in [4]). They are in the full version of the paper [5].

Let P be a multiprogram that uses READS, WRITES, BCASTS, and for each process, only calls DELIVER from at most one thread.

Theorem 4.11. $(\text{TS}_{\text{NW}}^{\text{POB}(L)}(P), \text{NW})$ implements $(P, \text{POB}(L))$ where DELIVER is called from at most one thread in each process.

Begin Proof

Assume: Let \widehat{C} be a computation in $\mathcal{C}(\text{TS}_{\text{NW}}^{\text{POB}(L)}(P), \text{NW})$ and let C be an interpretation of \widehat{C} .

To show $\text{POB}(L)[C]$ we construct witness orders that satisfy the requirements of Definition 2.3.

Build: Choose some collection of witness orders $\langle (O|\widehat{p}, \xrightarrow{\widehat{L}_{\widehat{p}}}) : \widehat{p} \in \text{TS}_{\text{NW}}^{\text{POB}(L)}(P) \rangle$ to $\text{NW}[\widehat{C}]$.

Recall that for a total order (O, \xrightarrow{X}) , its induced sequence is denoted 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 $(\widehat{O}|\widehat{p}, \xrightarrow{\widehat{L}_{\widehat{p}}})$ to $\text{NW}[\widehat{C}]$ and form $Short(\widehat{L}_{\widehat{p}})$ by removing:
 1. all the operations on the T , *counter*, and *local-counter* variables.
 2. all `send` and `recv` operations except: the `send` of a `[LOCAL-BROADCAST-REQUEST]`.
 3. all priority queue operations except `extractmin` operations.
 4. all fifo queue operations except `dequeue` operations.

Note that each remaining operation was produced from the transformation of some specification level operation, rather than a thread created by the transformation.

- Associate the target level operations in $Short(\widehat{L}_{\widehat{p}})$ with specification level ones by defining $\sim_{\text{construct}}$ as follows:

For this proof we just have to associate these target level operations with the specification level operations that produced them.

Target Operation	in transformation	Let
$\frac{\text{read}(x)}{v}$	$\text{TS}_{\text{NW}}^{\text{POB}(L)}(\frac{\text{READ}(x)}{v})$	$\frac{\text{read}(x)}{v} \sim_{\text{construct}} \frac{\text{READ}(x)}{v}$.
$\text{write}(x, v)$	$\text{TS}_{\text{NW}}^{\text{POB}(L)}(\text{WRITE}(x, v))$	$\text{write}(x, v) \sim_{\text{construct}} \text{WRITE}(x, v)$.
$\text{send}(s, d, [\text{L-B-R}])$	$\text{TS}_{\text{NW}}^{\text{POB}(L)}(\text{BCAST}(u, l))$	$\text{send}(s, d, [\text{L-B-R}]) \sim_{\text{construct}} \text{BCAST}(u, l)$.
$\frac{\text{extractmin}(queue)}{m}$	$\text{TS}_{\text{NW}}^{\text{POB}(L)}(\frac{\text{DELIVER}()}{m.update, m.label})$	$\frac{\text{extractmin}(queue)}{m} \sim_{\text{construct}} \frac{\text{DELIVER}()}{m.update, m.label}$.
$\frac{\text{dequeue}(fifo[p])}{m}$	$\text{TS}_{\text{NW}}^{\text{POB}(L)}(\frac{\text{DELIVER}()}{m.update, m.label})$	$\frac{\text{dequeue}(fifo[p])}{m} \sim_{\text{construct}} \frac{\text{DELIVER}()}{m.update, m.label}$.

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

Verify: The construction yields total orders $(O|p, \xrightarrow{L_p})$ that are valid by the fact that projecting on to subsets of objects preserves validity of the variables. The sequences of `BCAST` and `DELIVER` are valid since

1) the `send` of the `BCAST` $\xrightarrow{\text{HappensBefore}}$ the `extractmin` of its `DELIVER` and 2) each `ORD-MSG` is sent only once to each processor, received only once at each processor and enqueued only once at each processor. 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 following table shows the properties we prove to verify the constraints of Definition 2.3:

Constraint	Lemma
$\forall p \in P : (O p, \xrightarrow{L_p})$ is a valid total order	By construction
$\forall p \in P : \text{Extends}[O p, \xrightarrow{L_p}, \xrightarrow{\text{prog}}]$	Lemma 4.15
$\forall p \in P : \text{Extends}[O p, \xrightarrow{L_p}, \xrightarrow{\text{delorder}}]$	Lemma 4.16
$\forall p, q \in P, l \in L : \text{Agree}[O \text{delivers}(l), \xrightarrow{L_p}, \xrightarrow{L_q}]$	Lemma 4.17

End Proof

We prove the required lemmas: two key technical lemmas, 4.12 and 4.13, and three lemmas 4.15-4.17 that glue these results to the theorem.

Lemma 4.12. *If $\text{send}(s_1, d_1, m_1), \text{send}(s_2, d_2, m_2) \in O$ so that $m_1.l \neq \perp$ and $m_2.l \neq \perp$ and $m_1.ts \leq m_2.ts$ and $\frac{\text{extractmin}(\text{queue}_h)}{m_2} \in O|p$ then $\text{pri-enqueue}(\text{queue}_g, m_1) \xrightarrow{\widehat{\text{prog}}} \frac{\text{extractmin}(\text{queue}_h)}{m_2}$*

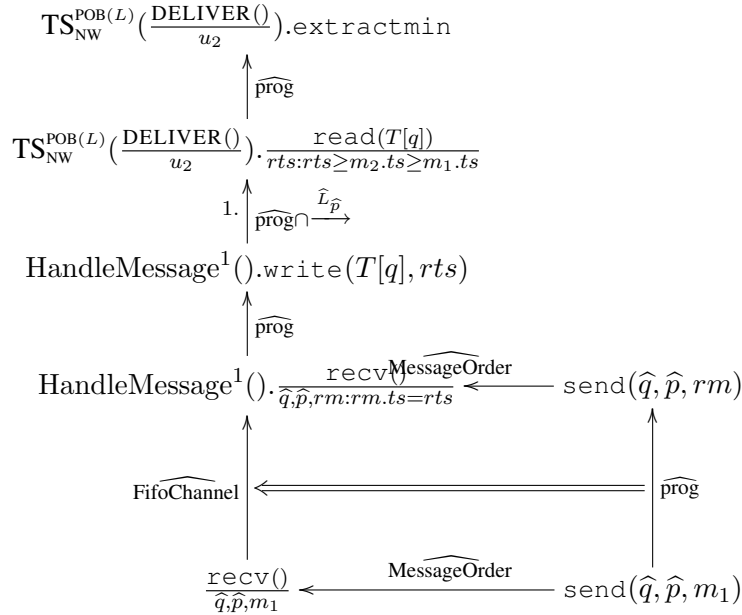
Begin Proof

Let m_1, m_2 be ORD-MSG messages so that $\text{send}(s_1, d_1, m_1), \text{send}(s_2, d_2, m_2) \in O$ and $m_1.ts \leq m_2.ts$ and $\frac{\text{extractmin}(\text{queue}_h)}{m_2} \in O|p$.

By examining the transformation $\frac{\text{extractmin}(\text{queue}_h)}{m_2}$ must come from the transformation of a $\frac{\text{DELIVER}()}{u_2}$. m_1 and m_2 have corresponding spec level updates u_1 and u_2 , respectively.

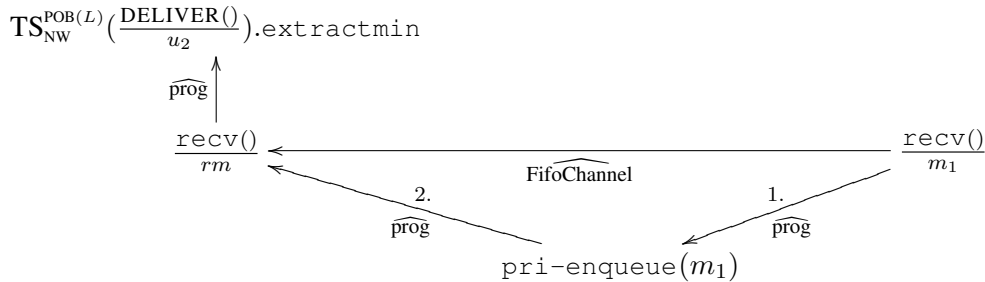
Let $q = m_1.src$.

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 $m_1 \neq rm$.



1. By the validity $\widehat{L}_{\widehat{p}}$ there must be a `write` that wrote that value. Since `deliver` is only in one thread, `HandleMessage` instances can only be in that thread, so they must be ordered in program order.

By examining the code, we can see that the `recv` of m_1 must have been followed by an `pri-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 `pri-enqueue` gives:



1. m_1 is a ORD-MSG. The HandleMessage instance that received m_1 will `pri-enqueue` it.
2. The `recv` operations must come from separate HandleMessage instances, which must be by the same thread.

Therefore $\text{pri-enqueue}(m_1) \xrightarrow{\widehat{\text{prog}}} \text{TS}_{\text{NW}}^{\text{POB}(L)}\left(\frac{\text{DELIVER}()}{u_2}\right) \cdot \frac{\text{extractmin}(queue_i)}{m_2}$.

End Proof

Lemma 4.12 is the essential safety property of the 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.

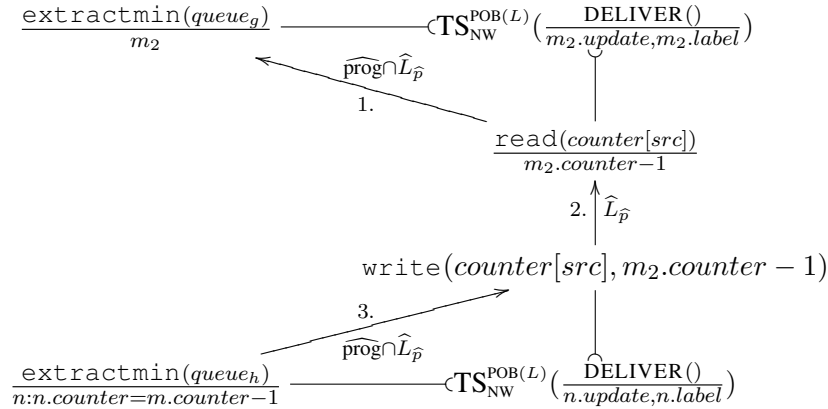
Lemma 4.13. *If m_2 is ORD-MSG sent by some $q \in P$ with $m_2.\text{counter} \geq 2$ then there is a ORD-MSG m_1 sent by the same q such that $m_1.\text{counter} = m_2.\text{counter} - 1$.*

If $d_1 = \frac{\text{extractmin}(queue_h)}{m_2}$ or $\frac{\text{dequeue}(fifo[q])}{m_2}$ $d_2 \in O|p$ for some $p \in P$ then $d_1 \xrightarrow{\widehat{\text{prog}}} d_2$ for some $d_1 = \frac{\text{extractmin}(queue_i)}{m_1}$ or $\frac{\text{dequeue}(fifo[q])}{m_1}$.

Begin Proof

m_1 must exist since the counter is incremented to a value c exactly when an ORD-MSG is created with that counter value c .

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



1. Examine the definition of $\text{TS}_{\text{NW}}^{\text{POB}(L)}\left(\frac{\text{DELIVER}()}{m,l}\right)$. The CanDeliver call before a `extractmin` invocation 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. This deliver is $\frac{\text{DELIVER}()}{n.update, n.label}$ for some message n .
3. Examine the definition of $\text{TS}_{\text{NW}}^{\text{POB}(L)}\left(\frac{\text{DELIVER}()}{n.update, n.label}\right)$ the only write to counter occurs right after a dequeue.

$n = m_1$ since different ORD-MSG messages from the same process have distinct counter values.

End Proof

Corollary 4.14. *If m_1, m_2 are ORD-MSG messages and $\hat{p} = m_1.\text{src} = m_2.\text{src}$ and $m_1.\text{counter} < m_2.\text{counter}$ then if $d_2 = \frac{\text{extractmin}(queue_h)}{m_2}$ or $\frac{\text{dequeue}(fifo[\hat{p}])}{m_2}$ and $d_2 \in O|p$ implies $d_1 \xrightarrow{\widehat{\text{prog}}} \frac{\text{extractmin}(queue_h)}{m_2}$ for some $d_1 = \frac{\text{extractmin}(queue_g)}{m_1}$ or $\frac{\text{dequeue}(fifo[\hat{p}])}{m_1}$.*

Begin Proof

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 $m_2.counter \geq 2$.

Apply Lemma 4.13 result inductively to get:

$$d_1 \xrightarrow{\widehat{\text{prog}}} d_2$$

$$\text{Where } d_1 = \frac{\text{extractmin}(queue_g)}{m_1} \text{ or } \frac{\text{dequeue}(fifo[\widehat{p}])}{m_1} \text{ and } d_2 = \frac{\text{extractmin}(queue_h)}{m_2} \text{ or } \frac{\text{dequeue}(fifo[\widehat{p}])}{m_2}.$$

End Proof

Lemma 4.15. $\forall p \in P : \text{Extends}[O|p, \xrightarrow{L_p}, \xrightarrow{\text{prog}}]$

Begin Proof

This proof is simplified by of the way transformations are defined, the memory model, and the choice of operations in the construction of the $\xrightarrow{L_p}$ orders.

Let $o_1, o_2 \in O|p$ such that $o_1 \xrightarrow{\text{prog}} o_2$. Let $\text{TKN}_{\text{NW}}^{\text{POB}(L)}(o).o$ be the operation in the transformation of o so that $\text{TKN}_{\text{NW}}^{\text{POB}(L)}(o).o \xrightarrow{\text{construct}} o$.

$$\begin{array}{ccc}
 o_1 & \xrightarrow{\text{prog}} & o_2 \\
 \text{TKN}_{\text{NW}}^{\text{POB}(L)} \downarrow & & \downarrow \text{TKN}_{\text{NW}}^{\text{POB}(L)} \\
 \text{TKN}_{\text{NW}}^{\text{POB}(L)}(o_1) & \xrightarrow{\widehat{\text{prog}}} & \text{TKN}_{\text{NW}}^{\text{POB}(L)}(o_2) \\
 \downarrow & & \downarrow \\
 \text{TKN}_{\text{NW}}^{\text{POB}(L)}(o_1).o & \xrightarrow[\text{1.}]{\widehat{\text{prog}} \cap \widehat{L}_{\widehat{p}}} & \text{TKN}_{\text{NW}}^{\text{POB}(L)}(o_2).o \\
 \text{construct} \downarrow & & \downarrow \text{construct} \\
 o_1 & \xrightarrow{L_p} & o_2
 \end{array}$$

1. $\text{Extends}[O|\widehat{p}, \xrightarrow{\widehat{L}_{\widehat{p}}}, \xrightarrow{\widehat{\text{prog}}}]$

End Proof

Lemma 4.16. $\forall p \in P : \text{Extends}[O|p, \xrightarrow{L_p}, \xrightarrow{\text{delorder}}]$

Begin Proof

Let $o_1, o_2 \in O|p$ such that $o_1 \xrightarrow{\text{delorder}} o_2$.

Then $o_1 = \frac{\text{DELIVER}()}{u_1, l_1}, o_2 = \frac{\text{DELIVER}()}{u_2, l_2}$ and $\text{BCAST}(u_1, l_2) \xrightarrow{\text{prog}} \text{BCAST}(u_2, l_2)$.

$$\begin{array}{ccc}
 \text{TKN}_{\text{NW}}^{\text{POB}(L)}(\text{BCAST}(u_1, l_1)).\text{send} & \xrightarrow{\widehat{\text{prog}}} & \text{TKN}_{\text{NW}}^{\text{POB}(L)}(\text{BCAST}(u_2, l_2)).\text{send} \\
 \downarrow \widehat{\text{MessageOrder}} & & \downarrow \widehat{\text{MessageOrder}} \\
 \text{HandleMessage}^1.\text{recv} & \xrightarrow{\widehat{\text{FifoChannel}}} & \text{HandleMessage}^2.\text{recv} \\
 \downarrow \widehat{\text{prog}} & \nearrow \widehat{\text{prog}} & \downarrow \widehat{\text{prog}} \\
 \text{HandleMessage}^1.\text{write}(\text{local-counter}, x) & & \text{HandleMessage}^2.\text{write}(\text{local-counter}, y) \\
 \downarrow \widehat{\text{prog}} & & \downarrow \widehat{\text{prog}} \\
 \text{HandleMessage}^1.\text{send}(m_1) & & \text{HandleMessage}^2.\text{send}(m_2)
 \end{array}$$

Since the *local-counter* monotonically increases, $0 < m_1.\text{counter} = x < y = m_2.\text{counter}$.

By Corollary 4.14 we have $\frac{\text{extractmin}(\text{queue}_g)}{m_1} \xrightarrow{\widehat{\text{prog}}} \frac{\text{extractmin}(\text{queue}_h)}{m_2}$ thus:

$$\frac{\text{extractmin}(\text{queue}_g)}{m_1} \xrightarrow{\widehat{\text{prog}} \cap \widehat{L}_{\widehat{p}}} \frac{\text{extractmin}(\text{queue}_h)}{m_2}$$

$$\begin{array}{ccc} \text{construct} \downarrow & & \text{construct} \downarrow \\ \frac{\text{DELIVER}()} {u_1, l_1} & \xrightarrow{L_p} & \frac{\text{DELIVER}()} {u_2, l_2} \end{array}$$

1. Extends $[O|\widehat{p}, \widehat{L}_{\widehat{p}}, \widehat{\text{prog}}]$
As required.

End Proof

Lemma 4.17. $\forall p, q \in P, l \in L : \text{Agree}[O|\text{delivers}(l), \xrightarrow{L_p}, \xrightarrow{L_q}]$

Begin Proof

Define $\text{match}(\frac{\text{DELIVER}()} {u, l}) = m : \text{TS}_{\text{NW}}^{\text{POB}(L)}(\frac{\text{DELIVER}()} {u, l}). \frac{\text{extractmin}(\text{queue}_g)}{m}$.

Let $l \in L$, define a total order $(O|\text{delivers}(l), \xrightarrow{\text{label}_l})$.

$$\frac{\text{DELIVER}()} {u_1, l} \xrightarrow{\text{label}_l} \frac{\text{DELIVER}()} {u_2, l} \stackrel{\text{def}}{=} (m_1.\text{ts}, m_1.\text{src}) < (m_2.\text{ts}, m_2.\text{src})$$

where $m_1 = \text{match}(\frac{\text{DELIVER}()} {u_1, l})$, $m_2 = \text{match}(\frac{\text{DELIVER}()} {u_2, l})$ and $<$ compares pairs lexicographically.

We now show that every $\xrightarrow{L_p}$ agrees with $\xrightarrow{\text{label}_l}$.

Let $\frac{\text{DELIVER}()} {u_1, l} \xrightarrow{\text{label}_l} \frac{\text{DELIVER}()} {u_2, l}$ and m_1, m_2 be their respective matches, then $(m_1.\text{ts}, m_1.\text{src}) < (m_2.\text{ts}, m_2.\text{src})$ and $m_1.\text{ts} \leq m_2.\text{ts}$ so by Lemma 4.12:

$$\begin{array}{ccc} \text{pri-enqueue}(\text{queue}_l, m_1) & \xrightarrow{\widehat{\text{prog}}} & \frac{\text{extractmin}(\text{queue}_l)}{m_2} \\ & \searrow \text{prog} & \uparrow \text{prog} \cap \widehat{L}_{\widehat{p}} \\ & & \frac{\text{extractmin}(\text{queue}_l)}{m_1} \\ & & \text{construct} \downarrow \\ & & \frac{\text{DELIVER}()} {u_1, l} \xrightarrow{L_p} \frac{\text{DELIVER}()} {u_2, l} \end{array}$$

1. Since the priority queue is ordered by lexicographic $<$ on $(m.\text{ts}, m.\text{src})$.
2. Extends $[O|\widehat{p}, \widehat{L}_{\widehat{p}}, \widehat{\text{prog}}]$

End Proof

5 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 $\text{safe}[x]$ and those that cannot by $\text{unsafe}[x] = \neg \text{safe}[x]$. The definitions of the models are in Section 2.

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

We evaluated these choices of partitions using the token and queue implementations described in Section 3. 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) .

$$\left(\text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}; \text{XyPOB}\right)(P, J) \stackrel{\text{def}}{=} \left(\text{XyPOB}(L(K))\right)\left(\text{WR}_{\text{POB}(L(K))}^{\text{PC}(K)}(P, J)\right)$$

	unoptimized	optimized
Single token	$\text{SWFR}_{\text{POB}(L(K_S))}^{\text{PC}(K_S)}; \text{TKN}_{\text{NW}}^{\text{POB}(L(K_S))}$	$\text{SWFR}_{\text{POB}(L(K_{SO}))}^{\text{PC}(K_{SO})}; \text{TKN}_{\text{NW}}^{\text{POB}(L(K_{SO}))}$
Multiple tokens	$\text{SWFR}_{\text{POB}(L(K_M))}^{\text{PC}(K_M)}; \text{TKN}_{\text{NW}}^{\text{POB}(L(K_M))}$	$\text{SWFR}_{\text{POB}(L(K_{MO}))}^{\text{PC}(K_{MO})}; \text{TKN}_{\text{NW}}^{\text{POB}(L(K_{MO}))}$
Single queue	$\text{SWFR}_{\text{POB}(L(K_S))}^{\text{PC}(K_S)}; \text{TS}_{\text{NW}}^{\text{POB}(L(K_S))}$	$\text{SWFR}_{\text{POB}(L(K_{SO}))}^{\text{PC}(K_{SO})}; \text{TS}_{\text{NW}}^{\text{POB}(L(K_{SO}))}$
Multiple queues	$\text{SWFR}_{\text{POB}(L(K_M))}^{\text{PC}(K_M)}; \text{TS}_{\text{NW}}^{\text{POB}(L(K_M))}$	$\text{SWFR}_{\text{POB}(L(K_{MO}))}^{\text{PC}(K_{MO})}; \text{TS}_{\text{NW}}^{\text{POB}(L(K_{MO}))}$

Only the $\text{SWFR}_{\text{POB}(L(K))}^{\text{PC}(K)}(i)$ implementation was tested as it was only realized later on that it was possible to implement FWSRtrans.

5.1 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 [7], 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  $\forall i \in 1, \dots, \text{large-array-length} : \text{large-array}[i] = \perp$  }
1 for  $i \leftarrow 1$  to test-repeats
2   do acquirelock(lock)
3     for  $j \leftarrow 1$  to large-array-length
4       do large-array[j] ← p
5     for  $j \leftarrow 1$  to large-array-length
6       do if large-array[j] ≠ p
7         then report-error()
8     releaselock(lock)

```

5.2 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 [12]. Recall that we assume processes are numbered starting at 0.

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

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

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

acquirelock($lock$)

```

1  repeat
2      while  $\exists q > p : lock . flag[q]$ 
3          do if  $lock . flag[p]$ 
4              then  $lock . flag[p] \leftarrow \text{FALSE}$ 
5           $lock . flag[p] \leftarrow \text{TRUE}$ 
6           $lock . turn \leftarrow \text{TRUE}$ 
7          while  $\exists q < p : lock . flag[q]$ 
8              do skip
          {in the until condition  $turn$  must be tested before  $flag$  }
9  until  $(lock . turn = p)$  and  $(\forall q > p : \neg lock . flag[q])$ 

```

releaselock($lock$)

```

1   $lock . flag[p] \leftarrow \text{FALSE}$ 

```

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:  $\forall i \in 1, \dots, m : lock[i]$ 
  - protected variables : safe  $\forall i \in 1, \dots, m : protected-variable[i]$  }
1  for  $i = 1$  to 300
2      do  $random-lock \leftarrow rand(\{1, \dots, m\})$ 
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, 8, and 9. 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.

5.2.1 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 10, 11, and 12. 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.

5.2.2 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 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 5 and 6, 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 13, 14, and 15. 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.

5.3 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()
{ Uses:
  – fake lock variables : unsafe  $\forall i \in 1, \dots, m : fake-lock-variable[i]$ 
  – protected variables : safe  $\forall i \in 1, \dots, m : protected-variable[i]$  }
1  for  $i = 1$  to  $\frac{300 \times |P|}{m}$ 
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

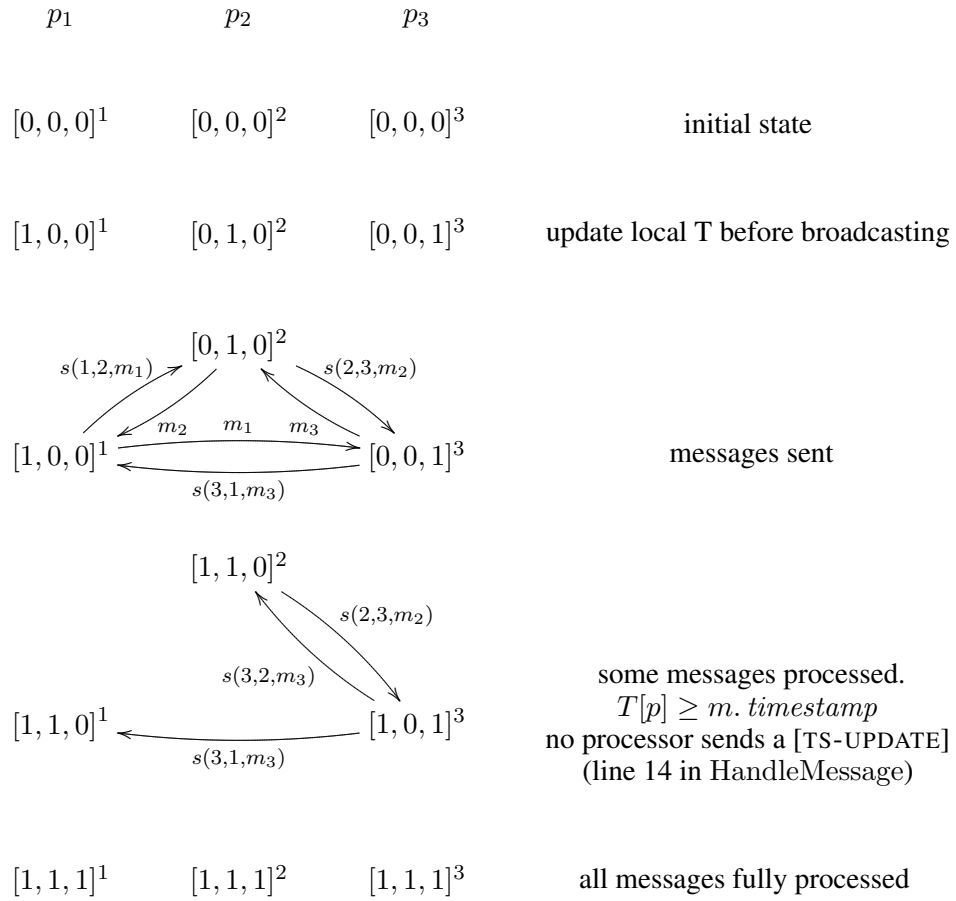


Figure 5: Example of best-case broadcast. Three messages broadcast using six messages.

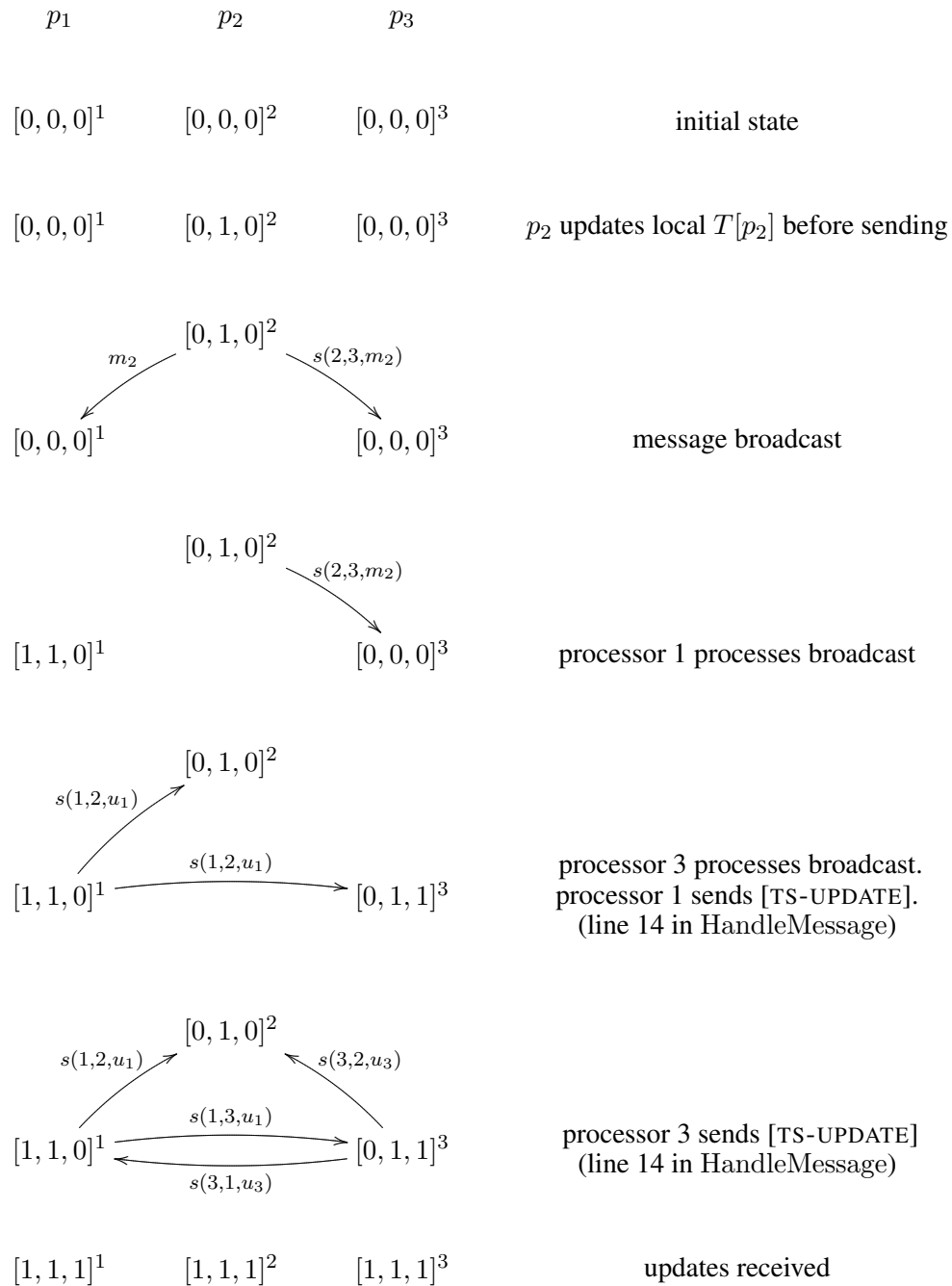


Figure 6: Example of worst-case broadcast. One message broadcast using six messages.

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.

5.4 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, $(\text{SWFR}_{\text{POB}(L(K_{SO}))}^{\text{PC}(K_{SO})}; \text{TKN}_{\text{NW}}^{\text{POB}(L(K_{SO}))})$), and its associated model $\text{PC}(K_{SO})$.

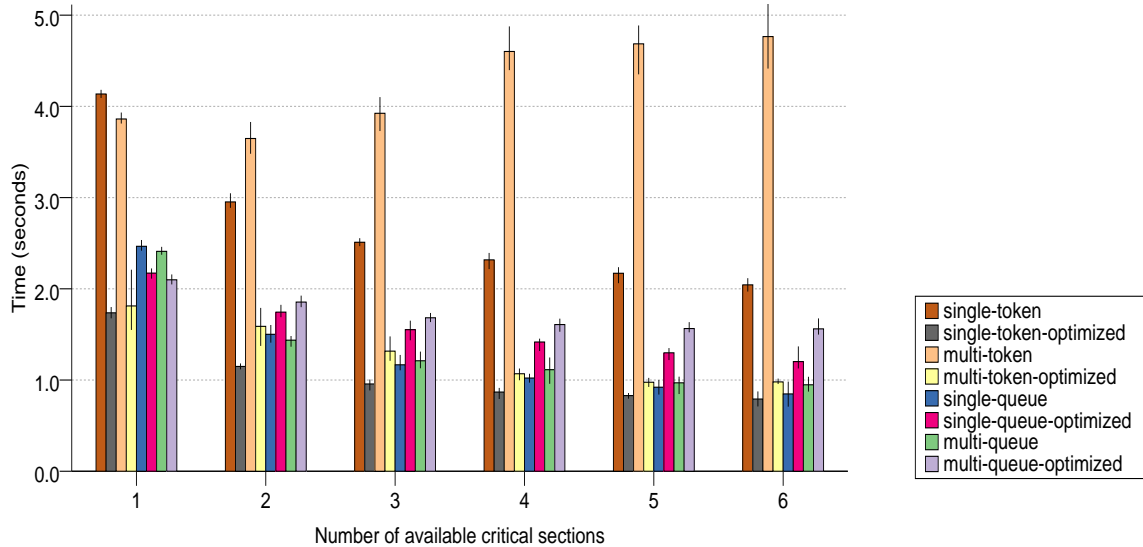


Figure 7: Total turnaround time for 8 processors, each performing 300 critical sections. Single token with optimization is the fastest. For both queue implementations, the optimized versions are slower.

6 Conclusion

- Introduced a general model, Partition Consistency
 - Introduced model of networked multithreaded processors with message passing
 - Introduced implementations of models
 - * Slow write/fast read and fast write/slow read implementations based on Attiya and Welch
 - * Queue based implementation extending Attiya and Welch total order broadcast, token based implementation
 - Proved implementations correct
 - Introduced model with broadcast primitive to clarify implementation structure
 - Implemented models and tested implementation performance

References

- [1] 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.
- [2] 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.
- [3] Hagit Attiya and Jennifer Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.
- [4] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2004.

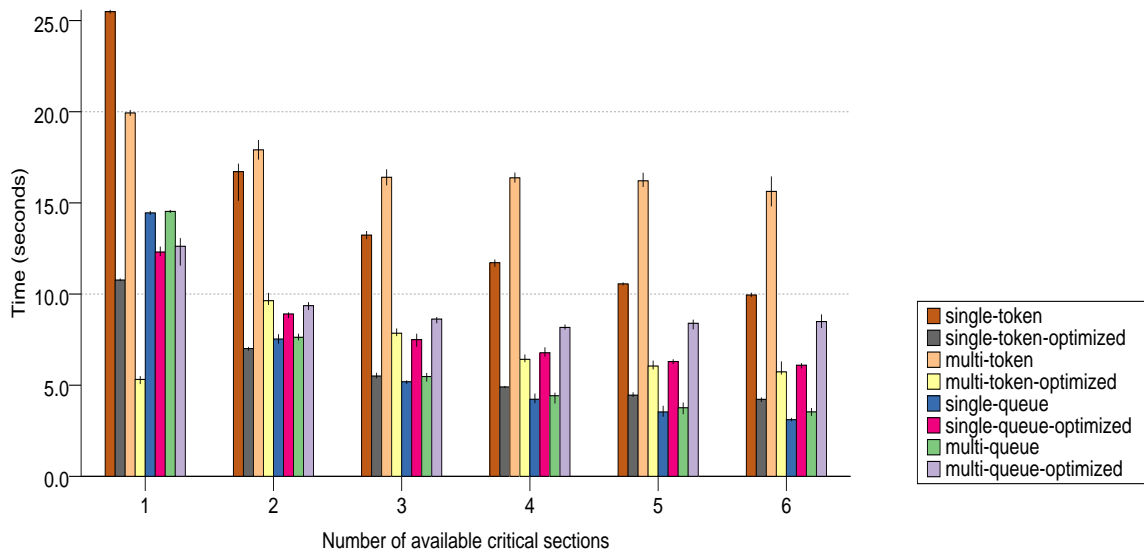


Figure 8: Total turnaround time for 16 processors, each performing 300 critical sections. Single queue is fastest implementation. Optimized queue implementations slower than regular queue implementations.

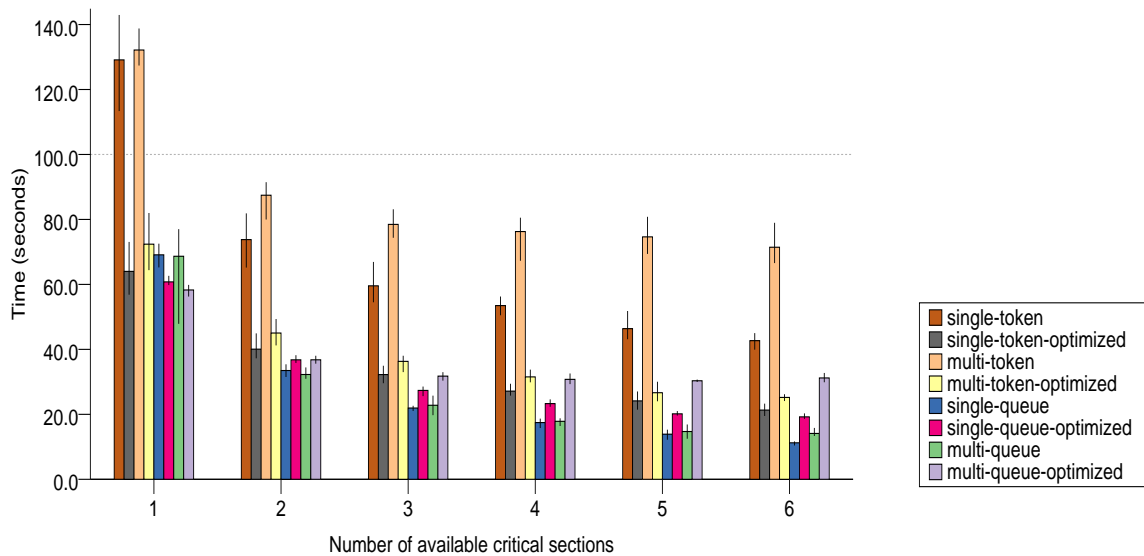


Figure 9: Total turnaround time for 24 processors, each performing 300 critical sections. Single queue is fastest implementation. Optimized queue implementations slower than regular queue implementations.

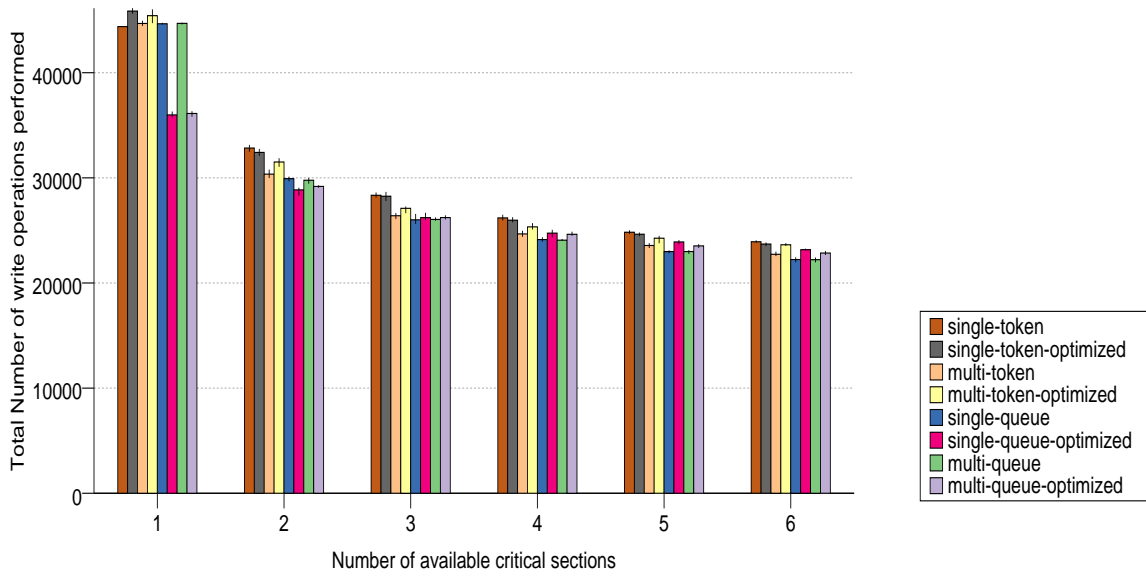


Figure 10: Total number of writes for 8 processors, each performing 300 critical sections.

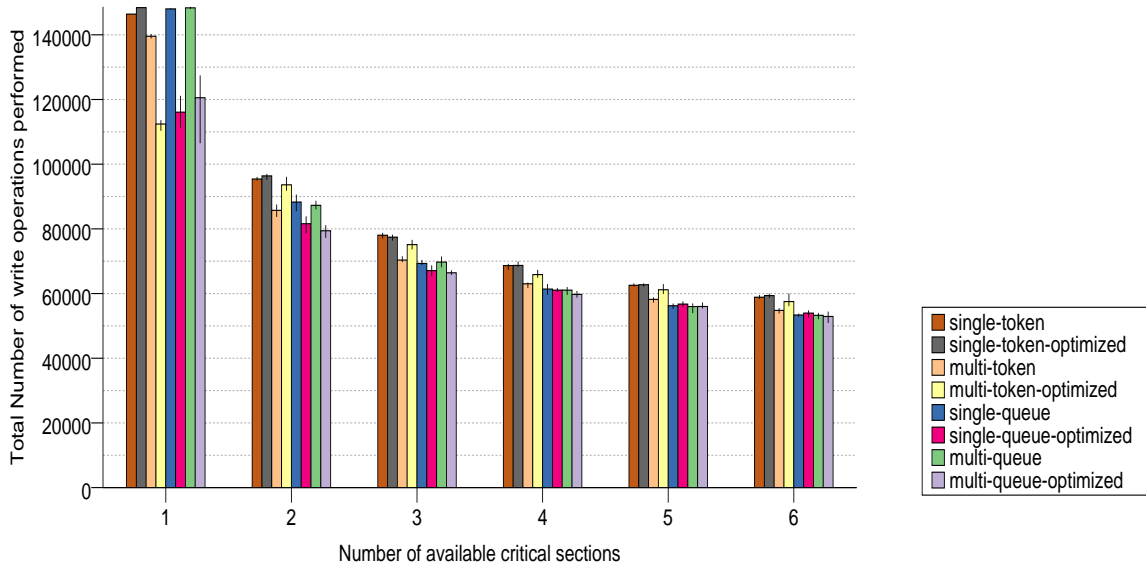


Figure 11: Total number of writes for 16 processors, each performing 300 critical sections.

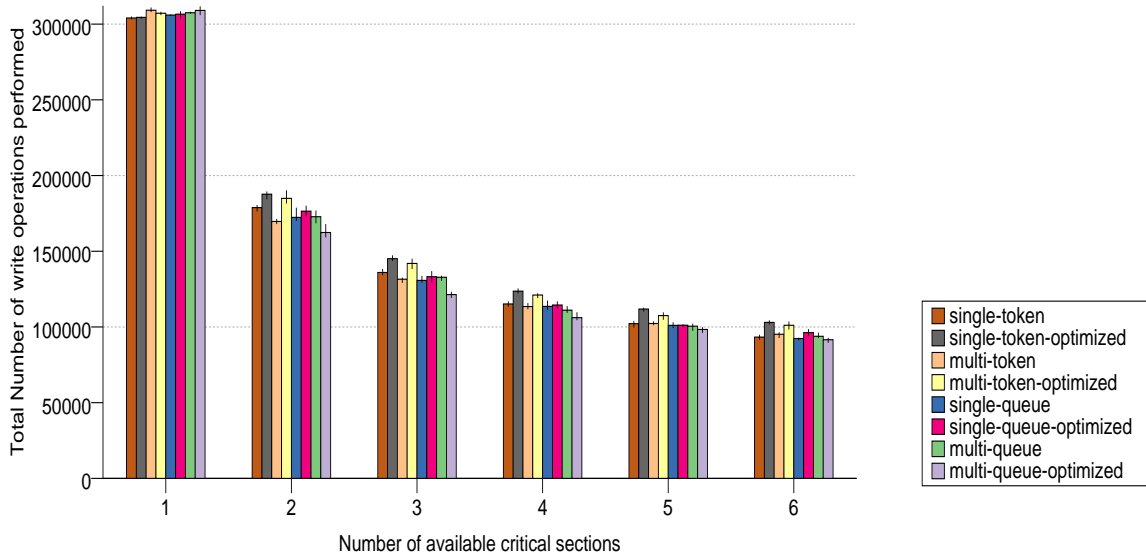


Figure 12: Total number of writes for 24 processors, each performing 300 critical sections.

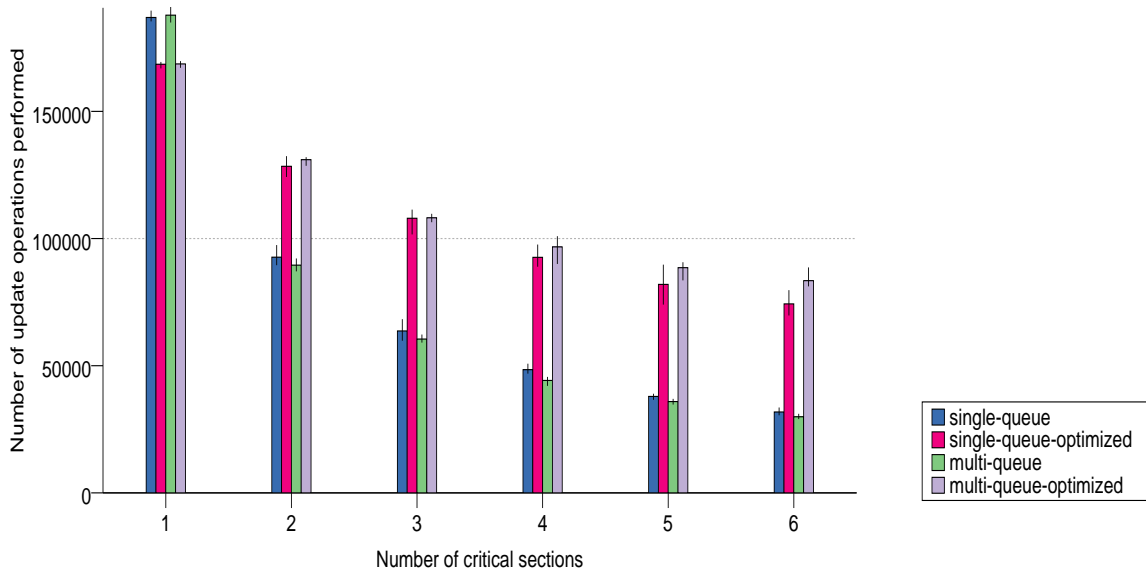


Figure 13: Total number of update operations for 8 processors, each performing 300 critical sections

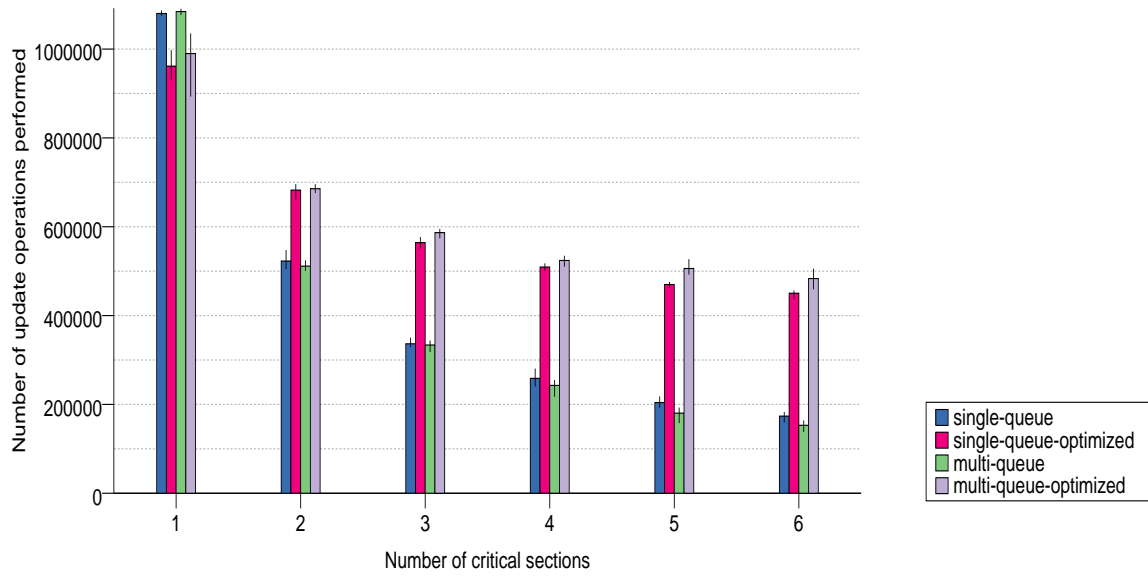


Figure 14: Total number of update operations for 16 processors, each performing 300 critical sections

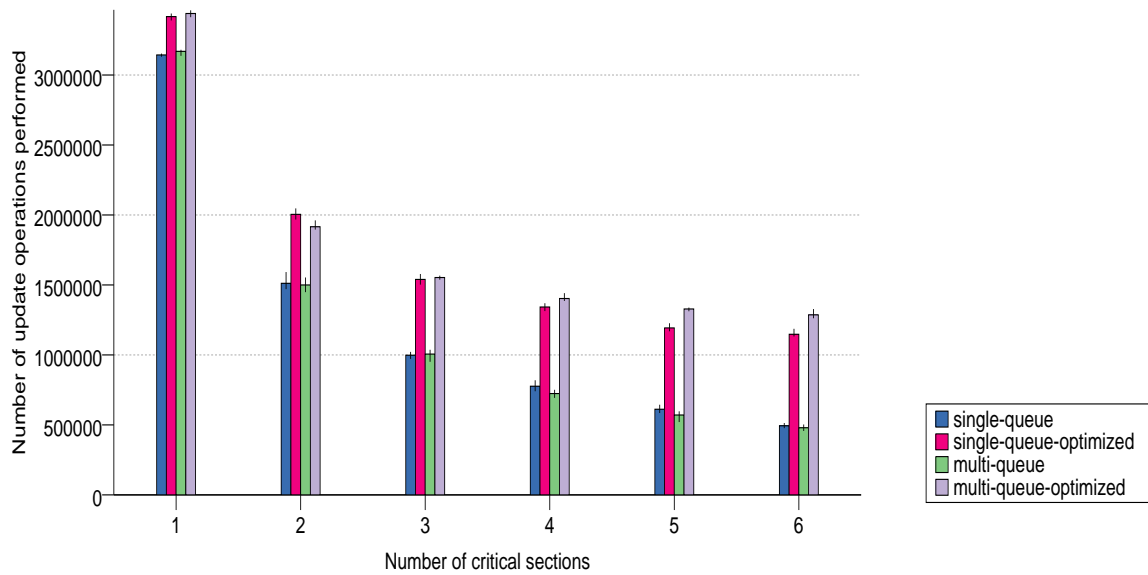


Figure 15: Total number of update operations for 24 processors, each performing 300 critical sections

- [5] Steven Cheng. Formal models and implementations of distributed shared memory. Technical Report 2009-922-01, University of Calgary, 2009.
- [6] 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.
- [7] Hewlett Packard Development Company. *HP MPI User’s Guide*. 2003. Order number: B6060-96022.
- [8] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [9] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [10] Lisa Higham, LillAnne Jackson, and Jalal Kawash. Specifying memory consistency of write buffer multiprocessors. *ACM Transactions on Computer Systems*, 25(1), 2007.
- [11] Lisa Higham, LillAnne Jackson, and Jalal Kawash. What is Itanium memory consistency from the programmer’s point of view? *Electr. Notes Theor. Comput. Sci.*, 174(9):63–84, 2007.
- [12] 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.
- [13] Intel Corporation. Intel Itanium architecture software developer’s manual, volume 2: System architecture. <http://www.intel.com/>, Oct 2002.
- [14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, July 1978.
- [15] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [16] Leslie Lamport. On interprocess communication (parts I and II). *Distributed Computing*, 1(2):77–85 and 86–101, 1986.
- [17] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.
- [18] Jeremy Manson, William Pugh, and Sarita Adve. The Java memory model. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 378–391. ACM, 2005.
- [19] 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.

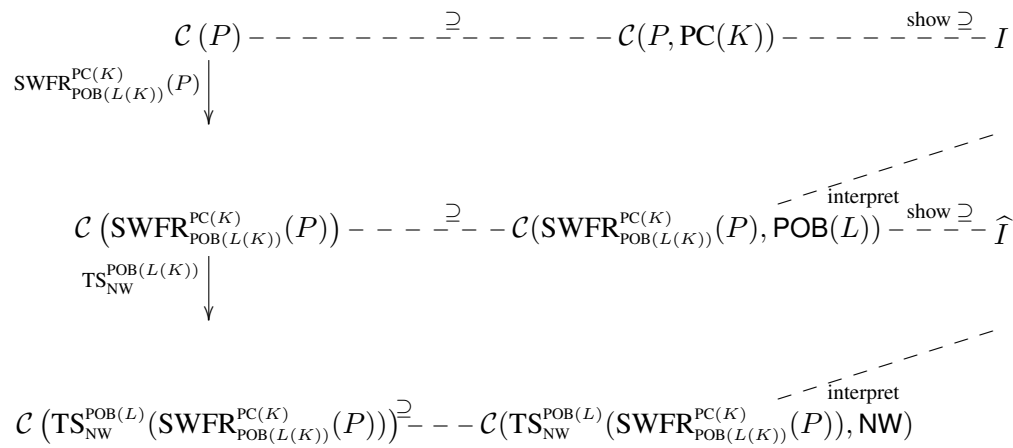


Figure 16: Proof obligation for establishing correctness of transformations