Abstracting consistency enforcement protocols with real-time orderings

Abstract

This work focuses on shared memory systems with a read-write interface (e.g., distributed datastores or multiprocessors). At the heart of such systems resides a protocol responsible for enforcing their consistency guarantees. Designing a protocol that correctly and minimally enforces consistency is a very challenging task. To address this challenge, we propose a set of mathematical abstractions, called real-time orderings (rt-orderings), that abstract the protocol. Then, we create a mapping from consistency guarantees to the rt-orderings and we relate the rt-orderings to protocol implementation techniques. Consequently, rt-orderings serve as an intermediate abstraction between consistency and protocol design, that allows designers to translate consistency guarantees into protocol implementations.

1 Introduction

This work focuses on "shared memory" systems that provide a read/write interface to the programmer. Such systems are ubiquitous in both the computer architecture and distributed systems. Prominent examples include shared memory multiprocessors (SMPs), GPUs, NoSQL Databases [1], coordination services[5], and software-based DSMs [6].

Such systems commonly replicate data – sometimes for performance (through caching), sometimes for fault tolerance and sometimes for both. To enable reasoning in the presence of replication, a *memory consistency model* (*MCM*) is specified as part of the system's interface, providing the rules that dictate what values a read can return. In order to enforce the MCM, the system deploys a protocol which ensures the replicas behave as the MCM dictates. This protocol is called *coherence protocol* in computer architecture and *replication protocol* in distributed systems. We simply refer to it as *the protocol*.

The MCM specifies the behaviour of the system when executing parallel programs. Specifically, it enumerates all patterns through which parallel programs can synchronize. For example, an MCM CM_a can provide guarantees on the synchronization pattern of Figure 1a (commonly known as "producer-consumer"). Specifically, in Figure 1a, process P1 writes object x and then y, while process P2 reads y and then x. CM_a enforces the guarantee that if P2's read to y returns the write of P1, then P2's read to x will also return the write of P1 (i.e., $x = 1$).

The MCM is thus a contract between the programmers and the designers. While programmers must understand the behaviour of the system, the designers must understand how to implement that behaviour. Problematically, enforcing the MCM is very challenging. For instance, consider

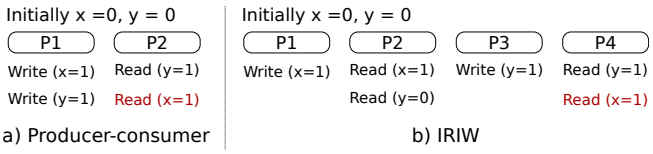

Figure 1: a) The producer-consumer synchronization pattern mandates that if P2 reads P1's write y, then it must also read P1's write to x. b) The independent-reads independent writes (IRIW) pattern mandates that if P2 sees P1's writes but not P3's and P4 does not see P3's write, then P3 must read the P1's write.

two well known MCMs: TSO [10] and Causal Consistency (CC) [11]. TSO enforces both Figure 1a and b while CC enforces Figure 1a but not b. Given this information, how is the designer to implement a correct and efficient protocol for either MCM? Crucially, how is the designer to differentiate between the two models? E.g., how to exploit that Figure 1b is not enforced in the CC protocol?

In this work we argue that we need a designer-centric abstraction which can act as an intermediary between the MCM and the protocol. That is: (1) we must be able to automatically translate any MCM to this abstraction; and (2) we must also be able to map the abstraction itself into protocol design choices.

To create this abstraction, we observe that a shared memory system, be it a multiprocessor or a geo-replicated Key-Value Store (KVS), can be abstracted through the model of Figure 2, which shows a bunch of processes executing a parallel program. Each process inserts its memory operations to a structure we call the *reorder buffer (ROB)*, allocating one ROB entry per operation. The *memory system* executes the operations it finds in the ROB and writes back the response of each operation in-place in its dedicated entry.

A process models a client of a KVS or a core of a multiprocessor. The ROB abstracts the core's load-store queue or a software queue that a KVS must maintain to keep track of incoming requests. The memory system is modelled as a distributed system comprising a set of *nodes*, where each node contains a controller and a memory. The nodes model the private caches of the multiprocessor or the geo-replicated memory servers of the KVS. Finally the network of the memory system can be thought of as the Network-on-Chip or as the WAN.

We observe that protocols of real systems enforce various consistency models using two widgets: 1) the ROB that allows for the reordering of operations; and 2) the memory system which determines how a read or a write executes, i.e., how it propagates to each of the replicas. We propose two sets of mathematical rules to abstract protocols, one that can abstract the reorderings of the ROB and one that can

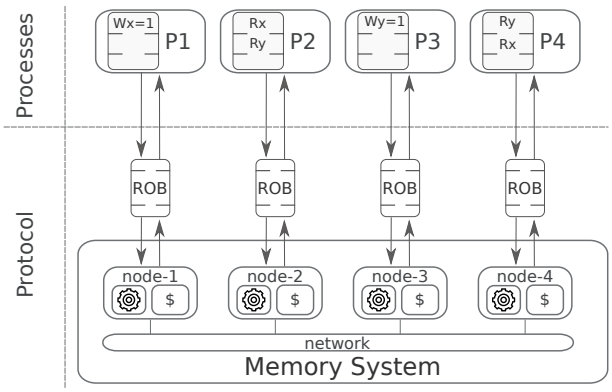

Figure 2: The model of a shared memory system. Processes P1-P4 execute the IRIW pattern of Figure 1b.

abstract the propagation rules of the memory system.

Specifically, we model the ROB using four rules named *program-order real-time orderings* (*prt-orderings*). The prt_{wr} prt-ordering mandates that when write w precedes read r in a program execution, then r can begin executing in the memory system only after w has completed. Similarly, we define $prt_{ww}, prt_{rr}, prt_{rw}$, for the rest of the combinations between reads and writes. To summarize, we model an ROB by specifying the subset of the four prt-orderings it enforces.

We model the memory system using four rules named *synchronization real-time orderings* (*srt-orderings*). The srt_{wr} srt-ordering mandates that if write w to object x completes before read r to object x in real time, then r must observe w. Similarly, we define $srt_{ww}, srt_{rr}, srt_{rw}$. To summarize we model a memory system by specifying the subset of the four srt-orderings it enforces.

We refer to prt-orderings and srt-orderings as *real-time orderings* (*rt-orderings*). The eight rt-orderings comprise our designer-centric intermediate abstraction of the protocol. Our key contribution is a mapping from MCMs to the rt-orderings. That is, given any MCM, we provide the set of minimal rt-orderings that enforce it. We present 16 such mappings in Table 3.

Using this mapping from MCMs to rt-orderings, we now revisit the questions we posed on Figure 1. Specifically, prt-orderings prt_{ww} and prt_{rr} and the srt-ordering srt_{wr} suffice to enforce the producer-consumer pattern (discussed in Section 5.4). Informally, prt_{ww} mandates that writes from the same process must be executed in the order intended by the program; prt_{rr} mandates the same for reads; srt_{wr} mandates that a read must return the value of the latest completed write (from any process). To also enforce the IRIW pattern, srt_{rr} is required (discussed in Appendix B). Informally, srt_{rr} mandates that if a read returns the value of write w, then any later read must also be able to return the value of write w.

Contributions. In summary, in this work we make the following contributions:

- We propose and formalize eight rt-orderings for mathematically modelling consistency enforcement protocols, serving as an intermediate abstraction between the MCM and the protocol implementation (§3).
- We create a mapping from MCMs to rt-orderings, specifying the minimal set of rt-orderings sufficient to enforce any MCM (§5).
- We informally map rt-orderings to protocol implementation techniques (§6).

2 Preliminaries

In this section, we first establish the system model, then we describe the notation we will use throughout the paper and finally we discuss how executions are modeled.

2.1 System Model

Figure 2 illustrates our system model. Specifically, there is a set of P *processes*, each of which executes a program and there is a *protocol* which enforces the MCM. The *protocol* is comprised of a set of *reorder buffers* (ROBs) and the *memory system*. The *memory system* stores a set X of shared *objects*, each of a unique name and value. A memory operation (or simply *operation*) can be either a read or a write of an object. A read returns a value and a write overwrites the value of the object and returns an ack.

ROBs. The ROBs are used to facilitate communication between the processes and the protocol. There are three events in the lifetime of an operation within the ROB: *issuing*, *begin* and *completion*. Specifically, we write that the process "issues an operation", when the operation is first inserted in the ROB. A process pushes operations in the ROB in the order encountered by its program. We refer to this order as the *program order*. Furthermore, we write that "an operation begins" when the memory system begins executing the operation. In this case the memory system marks the operation's ROB entry as *beginned*. Similarly, we write that "an operation completes" when the memory system has finished executing the operation. In this case, the memory system marks the operation's ROB entry as *completed* (and writes back the result if it is a read).

Notably, a process can use the value returned by a read, as soon as the read is marked *completed* by the memory system, irrespective of whether preceding operations are completed. An operation is removed from the ROB, if all three conditions hold: 1) it is the oldest operation, 2) it is completed and 3) the process has consumed its value, if it is a read.

Memory system. We model the memory system as a distributed system comprised of a set of *nodes*, where each node contains a controller and a memory. Furthermore, each node is connected to one ROB, from which it reads the operations that must be executed. The controller of each node is responsible for the execution of the operations. The memory of each node stores every object in X.

2.2 Notation

The notation used throughout this paper is the one used by Alglave et al. [2]. We repeat it here for completeness. The notation is based on relations. Specifically, we will denote the transitive closure of r with r^* and the composition of $r1$, $r2$ with $r1; r2$. We say that r is *acyclic* if its transitive closure is irreflexive and we denote by writing *acyclic(r)*. Finally, we say that r is a *partial order*, when r is transitive and irreflexive. We say that a partial order r is a *total order* over a set S, if for every x, y in S, it is that $(x, y) \in r \vee (y, x) \in r$.

We use small letters to refer to memory operations (e.g., a, b, c etc.). In figures that show executions (e.g., Figure 3), we denote a write with $Wx = val$ where x is the object to be written and val is the new value. Similarly, reads are represented as $Rx = val$, where val is the value returned. When required relations between operations are denoted with red arrows in figures.

2.3 Modelling executions

To model executions, we use the framework created by Alglave et al. [2], introducing minor changes where necessary. Specifically, we model an execution as a $tuple(M, po, rf, hb, RL)$. M is the set of memory operations included in the execution, po, rf and hb are relations over the operations and RL is a set of relations rl over the operations.

Execution relations (rf, hb and rl). The *program order po* relation is a total order over the operations issued by a single process, specifying the order in which memory operations are ordered in the program executed by the process. Operations are issued by a process in this order. Only operations from the same process are ordered. For two operations a, b from process $i \in \phi$, if a is issued before b then $(a, b) \in po$.

The *reads-from* rf relation contains a pair for each read operation in M, relating it with a write on the same object. For the pair $(a, b) \in rf$, it must be that a is a read that returns the value created by the write b, i.e., *a reads-from b*.

Finally, *happens-before hb* relation is a partial order over all operations, specifying the real-time relation of operations. Specifically, for two operations a, b, if $(a, b) \in hb$, then that means that a completes before b begins.

The *read-legal rl* relation is a total order over all operations, with the restriction that $acyclic(rl \cup rf)$. Given the rf of an execution E, it is often possible to construct more than one rl. We associate each execution E, with a set RL which contains every rl that can be constructed from E. For each $rl \in RL$, we derive the following relations.

Relations derived from rl (ws, fr, syn). The *write-serialization ws* relation is a total order of writes to the same object that can be derived from rl, such that $ws \subseteq rl$. Intuitively, ws captures the order in which writes to the same object serialize.

The *from-reads* fr relation connects a read to writes from the same object that precede the read in rl. Specifically, for every read a and a write b to the same object where $(a, b) \in rl$, then $(a, b) \in fr$. It follows that fr is transitive and $fr \subseteq rl$.

Finally, the *synchronization syn* relation combines the rf, fr and ws relations. Specifically, syn is a partial order over operations on the same object defined as the transitive closure of the union of rf, ws and fr, i.e., $syn \triangleq (rf \cup ws \cup fr)^*$. Two operations on the same object are said to synchronize if one of them is a write. Formally, for any pair of operations (a, b) on the same object, if at least one of a, b is a write then it must be that $(a, b) \in syn \vee (b, a) \in syn$. If both a, b are reads, then $(a, b) \in syn$ iff there exists a write c, such that $(a, c) \in syn \wedge (c, b) \in syn$.

Helper relations (po-type, syn-type, hb-type). Given the set of operations M, we define four relations WW, WR, RR, RW over M that contain all write-write, write-read, read-read and read-write pairs found in M, respectively. Table 2 uses these four relations to define the *po-type*, *syn-type* and *hb-type* relations. Specifically, by taking the intersection of po with each of WW, WR, RR, RW, we define $po_{ww}, po_{wr}, po_{rr}, po_{rw}$. We write *po-type* as a placeholder that can be replaced by any of these four relations. Notably, every pair in po is also in one of the *po-type* relations. I.e.,

$$po \triangleq po_{ww} \cup po_{wr} \cup po_{rr} \cup po_{rw}$$

Similarly, we define syn_{wr} and syn_{rr}. Note that we do not need to define syn_{ww} and syn_{rw}, because they would be the same as ws and fr, respectively. We write *syn-type* as a placeholder for $ws, syn_{wr}, syn_{rr}, fr$. We note that every pair in syn is also in one of the *syn-type* relations and that rf is a subset of syn_{wr}.

$$syn \triangleq ws \cup syn_{wr} \cup syn_{rr} \cup fr$$
$$rf \subseteq syn_{wr}$$

Finally, in the same spirit, we define the *hb-type* relations: $hb_{ww}, hb_{ww}, hb_{rr}, hb_{rw}$.

3 Real-time orderings

In this section we introduce and formally define the *real-time orderings* (rt-orderings), through which we model the protocol. There are two types of rt-orderings: the program-order real-time orderings (*prt-ordering*) through which we model the operation of the ROB and the synchronization real-time orderings (*srt-ordering*), through which we model the operation of the memory system. Table 1 provides the definition of each of the eight rt-orderings. Below we first describe the prt-orderings and then the srt-orderings.

3.1 Program-order Real-time Orderings

An execution $E(M, po, rf, hb, RL)$ is said to enforce the prt_{wr} if:

$$\forall w, r \in M \; s.t. \; (w, r) \in po_{wr} \rightarrow (w, r) \in hb_{wr}$$

prt-orderings		srt-orderings	
prt_{ww}	$po_{ww} \subseteq hb_{ww}$	srt_{ww}	$acyclic(hb_{ww} \cup syn)$
prt_{wr}	$po_{wr} \subseteq hb_{wr}$	srt_{wr}	$acyclic(hb_{wr} \cup syn)$
prt_{rr}	$po_{rr} \subseteq hb_{rr}$	srt_{rr}	$acyclic(hb_{rr} \cup syn)$
prt_{rw}	$po_{rw} \subseteq hb_{rw}$	srt_{rw}	$acyclic(hb_{rw} \cup syn)$

Table 1: The condition required to enforce each rt-ordering.

po-type	syn-type	hb -type
$po_{ww} \triangleq po \cap WW$	$ws \triangleq syn \cap WW$	$hb_{ww} \triangleq hb \cap WW$
$po_{wr} \triangleq po \cap WR$	$syn_{wr} \triangleq syn \cap WR$	$hb_{wr} \triangleq hb \cap WR$
$po_{rr} \triangleq po \cap RR$	$syn_{rr} \triangleq syn \cap RR$	$hb_{rr} \triangleq hb \cap RR$
$po_{rw} \triangleq po \cap RW$	$fr \triangleq syn \cap RW$	$hb_{rw} \triangleq hb \cap RW$

Table 2: The types of po, syn and hb relations

Plainly, a read r cannot begin before all writes that precede it in program order have completed. Table 1 extends this definition to write-write, read-read and read-write pairs. When all four prt-orderings are enforced then it follows that $po \subseteq hb$.

3.2 Synchronization Real-time Orderings

The synchronization real-time orderings (srt-orderings) are constraints over operations to the same object. There are four srt-orderings: $srt_{ww}, srt_{wr}, srt_{rr}, srt_{wr}$. An execution $E(M, po, rf, hb, RL)$ enforces the srt-ordering srt_{wr} if there exists $rl \in RL$ such that:

$$acyclic(hb_{wr} \cup syn)$$

In other words the srt_{wr} mandates that for a write a and a read b if $(a, b) \in hb_{wr}$, then it must be that $(b, a) \notin syn$. Table 1 describes the condition required to enforce each of the srt-orderings. A memory system is said to enforce an srt-ordering, if that srt-ordering is enforced in every execution that can be generated by the memory system. Notably, a combination of the srt-orderings can be enforced, for example srt_{ww}, srt_{wr} are both enforced when $acyclic(syn \cup hb_{ww} \cup hb_{wr})$. When all four srt-orderings are enforced then linearizability is enforced [4] (we expand on this on Appendix A).

4 Memory consistency models (MCMs)

In order to create a mapping from memory consistency models (MCMs) to the rt-orderings we need to establish a formalism for the MCMs. In this section, we use the formalism of Alglave et al. [2] to describe *synchronization patterns* (sync-pats) and assert that any MCM CM_i is defined as a set of sync-pats S_{CM}. An execution enforces CM iff it enforces every sync-pat in S_{CM}. Below we define what a sync-pat is and what its enforcement entails.

A sync-pat is a path between two operations to the same object. The path can be constructed through any composition of *po-type* and *syn-type* relations, with the only restriction being that at least one *po-type* must be included (explained in the remarks below). Table 2 describes which relations are denoted *po-type* and *syn-type*.

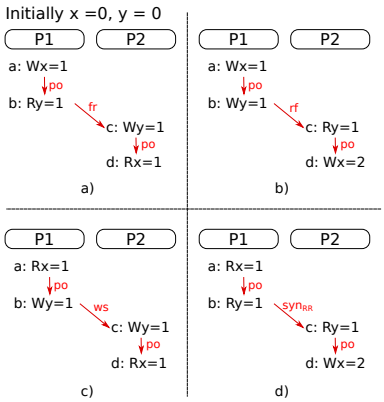

Figure 3: Four examples of sync-pats

For example, consider the sync-pat s that consists of a po_{wr} relation, then an fr relation and then a po_{wr} (depicted in Figure 3a). We can describe s by composing the three relations as follows:

$$s \triangleq po_{wr}; fr; po_{wr}$$

Given an execution $E(M, po, rf, hb, RL)$, we assert that E *enforces* s if there exists $rl \in RL$ from which we can derive a syn such that:

$$acyclic(s \cup syn)$$

In other words, a sync-pat that starts from operation a and ends on operation b is said to be *enforced* if $(b, a) \notin syn$.

Figure 3 depicts four sync-pats to serve as examples. In the execution of Figure 3a, the s sync-pat occurs between operations a and d. In this instance, s is enforced because d returns the value created by a. If d were to return 0, that would mean that there is a syn relation between d and a and thus s is not enforced.

Remarks. Note the following two remarks. First, we use syn rather than rl to test whether a sync-pat is enforced. This is because the sync-pat is a path between two operations to the same object, but rl is a total order of all operations across all objects. Second, note that a sync-pat must have at least one *po-type*, because otherwise it would just be a composition of *syn-type* relations. Any such composition is a subset of syn, which means that the execution enforces it by definition.

Consistency models. An MCM CM_i is defined by asserting that a set of sync-pats S_{CM} must be enforced. Plainly, an execution enforces CM_i iff it enforces every $s \in S_{CM}$ As an example, assume that CM_i is defined through the four sync-pats depicted in Figure 3a, which are formalized as follows:

$$s_1 \triangleq po_{wr}; fr; po_{wr},\ s_2 \triangleq po_{ww}; rf; po_{rw},$$
$$s_3 \triangleq po_{rw}; ws; po_{wr},\ s_4 \triangleq po_{rr}; syn_{rr}; po_{rw}$$

We define CM_i to be the following rule:

$$acyclic(s_1 \cup syn) \ \wedge \ acyclic(s_2 \cup syn)$$
$$\wedge\ acyclic(s_3 \cup syn) \ \wedge \ acyclic(s_4 \cup syn)$$

5 From MCMs to rt-orderings

In this section, we provide a mapping from MCMs to rt-orderings. Specifically, given an MCM CM_i that is specified through a set S_{CM} of sync-pats, we automatically infer a set of rt-orderings, sufficient to enforce CM_i. In the rest of this section, we first differentiate between regular and irregular sync-pats (§5.1). Then we provide the mapping (§5.2), prove its correctness for an example sync-pat (§5.3) and finally we extend the proof to all regular sync-pats (§5.4). We extend the proof to irregular sync-pats in Appendix B. In Table 3, we provide the mapping of 16 sync-pats to rt-orderings.

5.1 Regular sync-pats

We first categorize sync-pats in two classes: *regular* and *irregular*. A regular sync-pat is 1) composed by alternating *po-type* and *syn-type* relations and 2) starts and ends on a *po-type*. Thus, any regular sync-pat s, is of the following form:

$$s \triangleq po\text{-}type; syn\text{-}type; po\text{-}type; \ldots syn\text{-}type; po\text{-}type$$

Any sync-pat that does not conform to regularity rules is *irregular*. In this section, we create a mapping from a regular sync-pat to the sufficient rt-orderings.

5.2 The mapping

We assert the following three conditions to enforce any regular sync-pat, assuming that an operation can be of type m or n where $m, n \in \{read, write\}$.

- *Cond-1*: For every *po-type* relation po_{mn} found in s, the corresponding prt-ordering prt_{mn} must be enforced. Plainly any *po-type* relation must also be an *hb* relation.
- *Cond-2*: For every *syn-type* relation syn_{mn}, if it is not an rf, then the *reverse* srt-ordering srt_{nm} must be enforced.
- *Cond-3*: Finally, if the first operation in s is of type m and the last of type n, the corresponding srt-ordering srt_{mn} must be enforced.

5.3 Proof for an example sync-pat

We will start by first proving that the three conditions are sufficient for the simple sync-pat, that is portrayed in Figure 3a. Then we will extend to any regular sync-pat. The simple sync-pat s is the following:

$$s \triangleq po_{wr}; fr; po_{wr}$$

For s the three conditions require the following rt-orderings:

- *Cond-1*: The prt_{wr} is required for both po_{wr} relations.
- *Cond-2*: The srt_{wr} for the fr (i.e., syn_{rw}) relation.

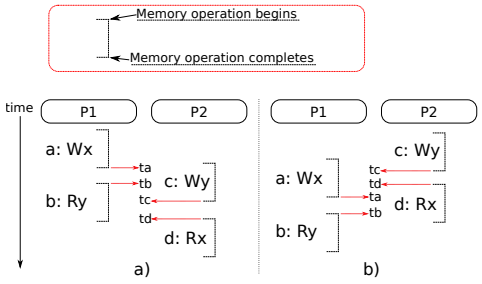

Figure 4: a) b begins before c completes b) c completes before b begins.

- *Cond-3*: The srt_{wr} because s starts with a write and ends on a read.

We want to show that for any execution $E(M, po, rf, hb, RL)$, if prt_{wr} is enforced and there exists $rl \in RL$ such that srt_{wr} is enforced then for the *syn* that is derived from that rl it holds that $acyclic(s \cup syn)$.

To prove this, we first establish a new relation named *begins-before-completes* (*bbc*). For two operations a, b we assert that if b does not complete before a begins (i.e., $(b, a) \notin hb$), then a begins before b completes (i.e., $(a, b) \in bbc$). We establish the following rule:

$$\forall a, d \in M \ s.t. \ (a, d) \in (hb; bbc; hb) \rightarrow (a, d) \in hb$$

We sketch a proof for this rule using Figure 4a . Figure 4a illustrates this through four operations (a, b, c, d), each of which is associated with a timestamp (ta, tb, tc, td). Operation b begins before c completes (i.e., $(b, c) \in bbc$), while also $(a, b), (c, d) \in hb$. Therefore, $(a, d) \in (hb; bbc; hb)$. Since a completes before b begins ($ta < tb$) and b begins before c completes ($tb < tc$), it follows that a completes before c completes ($ta < tc$). Because c completes before d begins ($tc < td$), it follows that a completes before d begins ($ta < td$). Therefore, $(a, d) \in hb$.

Figure 4b illustrates the counter-example, where c completes before b begins, in that case it is possible for d to begin before a completes.

Consider an execution $E(M, po, rf, hb, RL)$, for which there exists an $rl \in RL$ such that all three conditions we asserted are satisfied. Assume four operations $a, b, c, d \in M$ such that $(a, b) \in po_{wr} \wedge (b, c) \in syn_{wr} \wedge (c, d) \in po_{wr}$. This implies that $(a, d) \in s$. Let us now assume that $(a, d) \in (hb; bbc; hb)$ and thus $(a, d) \in hb$. Specifically, $(a, d) \in hb_{wr}$. This is illustrated in Figure 4a. From the cond-3 (srt_{wr}), we can assert that since $(a, d) \in hb_{wr}$ it follows that $(d, a) \notin syn$ and thus it must be that $acyclic(s \cup syn)$. Plainly, cond-3 ensures that $acyclic(s \cup syn)$ if the following condition holds:

$$\forall a, d \in M \ s.t. \ (a, d) \in s \rightarrow (a, d) \in hb$$

Therefore it suffices to prove that cond-1 and cond-2 guarantee the above condition. We will do so by proving that $(a, d) \in (hb; bbc; hb)$.

Firstly, cond-1 (prt_{wr}) mandates that $po_{wr} \subseteq hb_{wr}$. There-

fore, $(a, b), (c, d) \in hb$. We need only prove that $(b, c) \in bbc$. Let's assume that $(b, c) \notin bbc$. It follows that $(c, b) \in hb$. Recall, that b is a write and c is read. Cond-2 mandates that srt_{wr} is enforced: since $(c, b) \in hb$ then $(b, c) \notin syn$. We have reached a contradiction, and therefore it must be that $(b, c) \in bbc$.

Figure 4b provides the counter example where $(b, c) \notin bbc$. In this case it cannot be that $(a, d) \in s$ because $(b, c) \in hb$ and therefore from cond-2 it follows that $(c, b) \in syn$. Plainly, the sync-pat s will not occur here because P1's read to y will observe P2's write to y.

As we have shown above, from cond-1 and cond-2 it follows that $(a, d) \in hb$ and thus from cond-3 it follows that $(a, d) \notin syn$. Therefore the three conditions are sufficient to enforce s.

5.4 Extending to all regular sync-pats

Note the intuition behind the three conditions. Cond-1 ensures that the sync-pat cannot occur unless every *po-type* relation is also an *hb* relation. Similarly, cond-2 ensures that every *syn-type* relation is also a *bbc* relation. As a regular sync-pat is composed by alternating *po-type* and *syn-type* relations, enforcing cond-1 and cond-2 ensures that the sync-pat can be expressed as a composition of alternating *hb* and *bbc* relations.

As we saw $hb; bbc; hb \subseteq hb$. By induction we can extend this rule for any arbitrary length sequence of alternating *hb* and *bbc* relations. Specifically:

$$\forall a, b \in M\, s.t.\ (a, b) \in hb; bbc; hb \ldots. hb; bbc; hb \rightarrow (a, b) \in hb$$

Therefore, cond-1 and cond-2 ensure that if $(a, b) \in s$ then $(a, b) \in hb$ for any regular sync-pat s. Finally, cond-3 ensures that the sync-pat is enforced, by ensuring that if $(a, b) \in hb$ then $(b, a) \notin syn$. This means that our three conditions can be used to enforce any regular sync-pat. We extend to irregular sync-pats in Appendix B.

Examples – Table 3. Table 3 depicts the sufficient rt-orderings for 16 sync-pats. Specifically, each cell represents a distinct sync-pat between operations a, b, c, d. For instance, the highlighted cell where $a = Wx$, $b = Wy$, $c = Ry$, $d = Rx$, corresponds to the sync-pat of Figure 3a.

Cond-2 exception: rf. Recall that cond-2 is not required for rf edges. This is because the purpose of cond-2 is to ensure that the *syn-type* relation is also a *bbc* relation. However, this is implied by the rf as it is impossible for a read r to read-from a write w if r completes before w begins. Plainly: $\forall w, r \in M\, s.t.\ (w, r) \in rf \rightarrow (w, r) \in bbc$.

Producer-consumer (Figure 1a). Let s_{pc} be the producer-consumer sync-pat of Figure 1a (discussed in the Introduction). We assert that $s_{pc} \triangleq po_{ww}; rf; po_{rr}$. To enforce s_{pc}, cond-1 requires prt-orderings prt_{ww}, prt_{rr}, cond-2 does not require any srt-ordering because the only *syn-type* is an rf and cond-3 requires the srt_{wr}.

	c = Wy d = Ry	c = Wy d = Wy	c = Ry d = Ry	c = Ry d = Wy
a = Wx b = Rx	srt_{wr}, prt_{wr}	srt_{ww}, srt_{wr}, prt_{wr}, prt_{ww}	srt_{wr}, srt_{rr}, prt_{wr}, prt_{rr}	srt_{ww}, srt_{rr}, prt_{wr}, prt_{rw}
a = Wx b = Wx	srt_{wr}, srt_{ww}, prt_{ww}, prt_{wr}	srt_{ww}, prt_{ww}	srt_{wr}, srt_{rw}, prt_{ww}, prt_{rr}	srt_{ww}, srt_{rw}, prt_{ww}, prt_{rw}
a = Rx b = Rx	srt_{rr}, srt_{wr}, prt_{rr}, prt_{wr}	$srt_{rw}, , srt_{wr}$, prt_{rr}, prt_{ww}	srt_{rr}, prt_{rr}	srt_{rw}, srt_{rr}, prt_{rr}, prt_{rw}
a = Rx b = Wx	srt_{rr}, srt_{ww}, prt_{rw}, prt_{wr}	$srt_{rw}, , srt_{ww}$, prt_{rw}, prt_{ww}	srt_{rr} , srt_{rw}, prt_{rw}, prt_{rr}	srt_{rw}, prt_{rw}

Table 3: The mapping of 16 sync-pats to rt-orderings. Each cell represents a sync-pat each of the form $s \triangleq$ *po-type*; *syn-type*; *po-type*, where the first *po-type* includes (a, b), the *syn-type* includes (b, c) and the second *po-type* includes (c, d). The highlighted cell corresponds to Figure 3a.

6 From rt-orderings to Protocols

This paper has so far relied on the premise that, unlike sync-pats, the rt-orderings are helpful to the designer of the protocol. To address this, in this section, we relate rt-orderings to some well-known protocol design techniques. We start with a brief discussion on the prt-orderings and then we focus on the srt-orderings.

6.1 Enforcing rt-orderings

Prt-orderings model the operation of the ROB specifying when the memory system can begin executing an operation. Upholding the prt-orderings is as simple as inspecting the state of the ROB. For instance, enforcing prt_{wr} implies that the memory system cannot begin executing a read r from process p, until every preceding write in the ROB is completed.

Srt-orderings models how the memory system executes each of the two operations. Below we discuss two common techniques that can be used to enforce srt-orderings 1) *overlap* and 2) *lockstep*.

Overlap. The srt-ordering srt_{mn} can be enforced simply by ensuring that operations of type m must overlap with operations of type n in a physical location. For instance, we can enforce srt_{wr}, by ensuring that a write is propagated to x nodes and a read queries y nodes, where $x + y > N$ and N is the number of nodes. Alternatively, both types of operations can "meet" in some centralized physical location (e.g., the directory for multiprocessors). To ensure all four srt-orderings and thus linearizability, both reads and writes must query y nodes to learn about completed operations and must broadcast their results to x nodes. This is exactly how the multi-writer variant of ABD [8] operates.

Lockstep. Lockstep is a technique, where a memory system node first "grabs a lock" on the object before beginning an operation and releases it when the operation completes. Upon grabbing the lock, the node learns about the operation executed by the previous lock holder. The act of "grabbing the lock" is similar to getting a cache-line in M or S state in

a coherence protocol [9], or proposing to become the leader of the next log entry in a state machine replication protocol, such as Paxos [7].

There are two aspects of lockstep that can enforce srt-orderings. First, the srt-ordering between two operations is enforced if a lock must be passed from one to the other. For example we can enforce srt_{ww} by mandating that a lock must be grabbed to perform a write. Second, locking also ensures that certain operations cannot overlap in real-time. When a write cannot overlap with a write and srt_{ww} is enforced then srt_{rw} is also enforced. This is because if a read r returns the value of write w, then a write k that begins after r has completed must also begin after w has completed. Similarly, when a write cannot overlap with a read and srt_{wr} is enforced, then srt_{rr} is enforced. This is because if a read r returns the value of write w, then it must be that w completes before r begins. Therefore, a read m that begins after r has completed, must also begin after w has completed and thus will observe w. Protocols often combine the two aspects of lock-step to enforce the single-writer multiple-reader invariant (SWMR) [9], where for any given object at any given time there is either a single write in progress or multiple reads. This ensures that writes must grab a lock from each other (srt_{ww}), reads must grab a lock from writes (srt_{rw}), writes cannot overlap in time (srt_{rw}) and writes do not overlap in time with reads (srt_{rr}).

7 Conclusion

In this work we focused on the protocols that are responsible to enforce consistency in shared memory systems. We argued for the need of an intermediate abstraction between the consistency model and the protocol implementation, that will allow the designer to map the consistency model to specific protocol implementation techniques. We created this abstraction by mathematically abstracting the protocol through eight rt-orderings. Specifically, we observed that any such protocol is comprised by two widgets: the ROB and the memory system. We mathematically abstracted the ROB with the four prt-orderings and the memory system with the four srt-orderings. Crucially, we created a mapping from consistency guarantees to the rt-orderings, such that any consistency model can be translated into the minimal set of rt-orderings that are required to enforce it. Finally, we completed the picture, by relating the rt-orderings to protocol implementation techniques.

References

[1] "Manhattan, our real-time, multi-tenant distributed database for twitter scale," https://bit.ly/2Vgu0wd, April 2014, (Accessed on 07/08/2019).

[2] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats," *ACM TOPLAS*, vol. 36, no. 2, pp. 1–74, jul 2014.

[3] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[4] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855851

[6] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, ser. WTEC'94. Berkeley, CA, USA: USENIX Association, 1994, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267074.1267084

[7] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[8] N. A. Lynch and A. A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, June 1997, pp. 272–281.

[9] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence, second edition," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 1, pp. 1–294, 2020. [Online]. Available: https://doi.org/10.2200/S00962ED2V01Y201910CAC049

[10] S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: X86-TSO," in *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 391–407. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03359-9_27

[11] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible Update Propagation for Weakly Consistent Replication," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 288–301. [Online]. Available: https://doi.org/10.1145/268998.266711

Appendices

A Relation of srt-orderings to linearizability

Formally, the lin criterion [4] can be written in our notation as follows. An execution $E(M, po, rf, hb, RL)$ is linearizable if there exists $rl \in RL$ such that $hb \subseteq rl$. Furthermore, because lin is a *local* property, it suffices that only operations to the same object abide by the rule. Plainly, for two operations a, b to the same object x, if $(a, b) \in hb$ then it must be that $(a, b) \in rl$. This is equivalent to: $acyclic(hb, syn)$. This, in turn, is equivalent to enforcing all four srt-orderings. Therefore, the lin property is the property of enforcing all four srt-orderings.

Intuition. Intuitively, the lin property can be defined by writing that *each memory operation appears to take effect at some point between its invocation and completion* [3]. Notably, each of the srt-orderings is a specialization of this definition. For instance, srt_{wr} mandates that each write appears to take effect at some point between its invocation and completion

w.r.t. every read. Combining all four srt-orderings mandates that each write or read appears to take effect at some point between its invocation and completion w.r.t. every read or write. This is equivalent to the lin definition.

B Mapping irregular sync-pats to rt-orderings

A sync-pat is deemed *irregular* if 1) it has consecutive *po-type* relations or 2) it has consecutive *syn-type* relations or 3) it does not start with a *po-type* relation or 4) it does not end with a *po-type* relation. For each such sync-pat s, we derive a sync-pat s' such that 1) s' is regular and 2) if s' is enforced then s must also be enforced.

We start with sync-pats with consecutive *po-type* relations. We use the insight that any composition of *po-type* relations must be the subset of one of $po_{ww}, po_{wr}, po_{rr}, po_{rw}$ relation. For instance $po_{ww}; po_{wr}; po_{rr}; po_{rw}$ is a subset of po_{ww}. Therefore, for any s that has one more consecutive composed *po-type* relations, we derive a s' which replaces them with a single *po-type* relation, such that $s \subseteq s'$. Below is an example of s and the derived s'.

$$s_{po} \triangleq po_{wr}; po_{rw}; syn_{wr}; po_{rr}$$
$$s'_{po} \triangleq po_{ww}; syn_{wr}; po_{rr}$$

using the insight that $(po_{wr}; po_{rw}) \subseteq po_{ww}$.

Note that an alternative approach would have been to omit deriving s' and instead simply asserting that the first condition is applied for each *po-type* relation in s. That would still be correct and can be used.

Our approach is identical for sync-pats with consecutive composed *syn-type* relations. Specifically, a composition of *syn-type* relations is always a subset of one of fr, ws, syn_{wr}, syn_{rw}. This is because any pair in a composition of *syn-type* relations is also in *syn* and thus it must be in one of $fr, ws, syn_{wr}syn_{rw}$ as *syn* is the union of these four relations. Therefore, for any s that has one more consecutive composed *syn-type* relations, we derive a s' which replaces them with a single *syn-type* relation, such that $s \subseteq s'$.

If a sync-pat s begins with a *syn-type* relation, then we derive s' by simply removing it. Let us see why through an example. Assume the following s and derived s':

$$s \triangleq fr; po_{wr}fr; po_{wr}$$
$$s' \triangleq po_{wr}fr; po_{wr}$$

Assume now that $(a, c) \in s$, $(b, c) \in s'$ and $(a, b) \in fr$. Let us now prove that if s' is enforced, s is enforced too. Assume s' is enforced but s is not. Therefore, $(c, b) \notin syn$ but $(c, a) \in syn$. We know that $(a, b) \in fr$ and thus $(a, b) \in syn$. By the transitivity property of *syn* we assert that since $(c, a) \in syn \wedge (a, b) \in syn$ it must be that $(c, b) \in syn$. This contradicts our assumption that s' is enforced. Therefore enforcing s' is sufficient to also enforce s.

Similarly, if s ends on a *syn-type* relation, we remove it to derive s'. Assume the following example.

$$s \triangleq po_{wr}fr; po_{wr}fr$$
$$s' \triangleq po_{wr}fr; po_{wr}$$

Assume now that $(a, c) \in s$, $(a, b) \in s'$ and $(b, c) \in fr$. Using the same proof as above, we can infer that if s' is enforced then $(b, a) \notin syn$ and thus it follows that $(c, a) \notin syn$. Therefore enforcing s' is sufficient to also enforce s.

IRIW (Figure 1b). Let s_{iriw} be the IRIW sync-pat of Figure 1b (discussed in the Introduction). We assert that $s_{iriw} \triangleq rf; po_{rr}; fr; rf; po_{rr}$. This is an irregular sync-pat. We use the rules above to derive $s'_{iriw} \triangleq po_{rr}; syn_{rr}po_{rr}$ and assert that if s'_{iriw} is enforced then s_{iriw} is also enforced. To enforce s'_{iriw}, cond-1 requires the prt-ordering prt_{rr} and cond-2 requires the srt-ordering srt_{rr} which is also required by cond-3.

C From rt-orderings to MCMs

So far we have created a mapping from MCMs to rt-orderings, by establishing the sufficient rt-orderings to enforce any sync-pat. In this section, we use this result to obtain the reverse mapping from rt-orderings to MCMs, focusing our discussion solely on regular sync-pats, seeing as the enforcement of irregular sync-pats is done by mapping them to regular ones. To obtain the mapping from rt-orderings to sync-pats, we reverse each of the three conditions, assuming that an operation can be of type m or n where $m, n \in \{read, write\}$. Specifically, a sync-pats is enforced when it abides by the following conditions:

- *Cond-1'* : s can include a *po-type* relation po_{mn} iff the corresponding prt-ordering prt_{mn} is enforced.
- *Cond-2'*: s can include a *syn-type* relation syn_{mn} iff it is an rf relation or the reverse srt-ordering srt_{nm} is enforced.
- Cond-3': s can start with an operation of type n and finish on an operation of type n, iff the corresponding srt-ordering srt_{mn} is enforced.

Example. To showcase how these conditions can be used in practice, we specify the MCM CM_i that is enforced by a protocol that enforces the rt-orderings $prt_{ww}, prt_{wr}, srt_{rw}, srt_{wr}$. We do so by specifying a set of sync-pats S_{CM} where each regular sync-pats $\in S_{CM}$ abides by the following rules:

- Any *po-type* relation in s is either a po_{ww} or a po_{wr}
- Any *syn-type* relation in s is either in syn_{wr} or in syn_{rw}. (rf is included in syn_{wr})
- s must either start on a write and end on a read, or start on a read and end on a write.

Notably, the interplay amongst the rules can be used to further simplify them. For instance the third rule asserts that from the available srt-orderings (srt_{rw} and srt_{wr}) it follows that either the first operation must be a read and the last a write or the reverse. However, neither of the available

po-types (po_{ww} and po_{wr}) start with a read and since a regular sync-pat must start with a *po-type* relation, it cannot be that the first operation is a read. Consequently, the first operation can only be a write, and thus the last operation must be a read, to abide by the third rule.

Similarly, because a regular sync-pat, is a a composition of alternating *po-type* and *syn-type* relations, it follows that the available *po-type* and *syn-type* relations can only be used if they can synergize. For instance, the syn_{wr} cannot be used at all because neither of the available *po-types* (po_{ww} and po_{wr}) starts from a read. Similarly, because the syn_{wr} cannot be used, the po_{ww} cannot be used before any *syn-type*, nor can it be used as the last relation because the sync-pat must end on a read.

Therefore the rules for a sync-pat $s \in S_{CM}$ are simplified as follows:

- Any *po-type* relation in s must be a po_{wr}
- Any *syn-type* relation in s must be a syn_{rw}.
- s must start on a write and end on a read.

As a result any regular sync-pat $s \in S_{CM}$ is a composition of alternating po_{wr} ans syn_{rw} relations. Below, we list a few examples sync-pats in S_{CM}

$$s_1 \triangleq po_{wr}; syn_{rw}; po_{wr}$$

$$s_2 \triangleq po_{wr}; syn_{rw}; po_{wr}; syn_{rw}; po_{wr}$$

$$s_3 \triangleq po_{wr}; syn_{rw}; po_{wr}; \ \ldots \ syn_{rw}; po_{wr}$$

Notably, enforcing the prt_{ww} and srt_{rw} are not contributing towards any of the enforced any $s \in S_{CM}$.