RECORD AND REPLAY UNDER RELAXED CONSISTENCY

BY

RUSSELL LLEWELLYN JONES

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Nitin Vaidya

ABSTRACT

In the area of debugging parallel executions, record and replay is a technique that allows deterministic debugging even in the presence of data races. It is useful as most programmers are used to re-executing programs to find bugs. However, very little is known about how the consistency model affects record and replay. Previous work only applied to very strong consistency models, or to a specific architecture of shared memory. Very little theoretical basis has been developed for record and replay. This thesis makes three contributions:

- An algorithm that records the minimum record for record and replay under causal consistency.
- A demonstration that guaranteeing progress for a given replay mechanism can depend on the consistency model.
- A demonstration that *heterogeneous consistency* record and replay is possible, that is, it is possible to record an execution on one consistency model and replay it on another.

To my parents, for their love and support; to Nitin, for guiding me through graduate school; and to Rakesh, for teaching me how to think about research.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Debugging multithreading programs is difficult. The largest challenge that multithreading introduces is non-determinism due to races in thread execution. There are several ways to deal with this non-determinism. One can avoid it by using software, hardware, or programming techniques that limit the degree of non-determinism. Also, during testing, one can use hardware and software systems that guarantee all non-deterministic paths are exercised and tested. Additionally, one can exercise non-functional execution paths during debugging to discover the problems in the code and fix them. These classes of techniques complement each other. The focus of this thesis is on record and replay, which is in the last class.

Record and Replay (RnR) allows non-deterministic parallel program debugging to proceed in the way most programmers are most familiar with: The programmer runs their program, and notices incorrect behavior. The programmer then reruns the program, while more closely watching program state (either through additional output or with a debugger), and discovers where the problem occurs. However, even when a program has the same input, a multithreaded program may behave in a different way when rerun, due to races in thread execution. Thus, the observed bug may not occur, or another bug might happen instead, making it quite difficult to discover the cause of the original problem. Record and replay fixes this problem by creating a *record* during the original execution that allows the RnR system to guarantee that the rerun executes the same way as the original execution. In other words, the original execution is still non-deterministic, but the rerun is not. The RnR system may be implemented in software, in hardware, or in both. This work investigates two questions: How does one do RnR in relaxed consistency models? And, how does one minimize the record for RnR?

Memory consistency is the answer to the question: How do memory operations interact? In the load/store memory model there is a simple answer

for the single process application: A load to a variable returns the value that was last stored to that variable. However, with parallel applications this rule falls apart, because multiple processes may try to store to a variable concurrently, or one process may not have communicated the stored value to a process by the time the latter process performs a load. There are many ways of resolving this, giving rise to many consistency models.

There are stricter models and less strict (more relaxed) models. In general, stricter models are conceptually more similar to the single process memory model and easier for programmers to use, whereas relaxed models have better performance especially where communication costs are high, such as shared memory that is distributed over the internet.

1.1 Formal Definitions

1.1.1 Distributed Shared Memory

This work will focus on read-write (load-store) distributed shared memory. That is, only two operations are available: $write(variable, value)$, which changes the value of the variable to *value*, and returns nothing; and $read(variable)$, which returns the value of the variable. The system is composed of a network of processes. All processes can communicate with all other processes. All writes are eventually received by all processes. Reads by a process are only observed by that process. Each process has its own copy of each variable, which may not match the other process's copy (depending on the consistency), but will always hold some written value or the initial value of that variable. There is no assumption about the specific implementation of the consistency model; however, we do assume that the consistency condition is always satisfied in both the original execution and the replay.

1.1.2 Operations

Two notations for operations will be used. The first is mostly for examples, due to its readability. The second will be used largely in the proofs, for its precision and flexibility.

In examples, $R_X(v)$ will refer to a read to variable X that returns v; $W_X(v)$ will refer to a write that stores v to variable X.

Similar to [1], in the definitions and proofs, we use a 4-tuple to refer to an operation: (r, i, x, n) refers to the nth operation by process i, which is a read (r) to variable x (this notation differs slightly from [1]). Similarly, (w, i, x, n) refers to the nth operation by process i, which is a write (w) to variable x. For sets of operations, we use $*$ as a wild card for any elements of the tuple (e.g. $(r, i, *, *)$ refers to all reads by process i).

1.1.3 Consistency

In any system with multiple processes accessing a shared memory, there must be a consistency model, which is a set of rules defining what executions are legal. An execution is a sequence of operations and the values that are read or written by those operations. The consistency model for a distributed system determines what executions are legal.

In order to define the consistency rules we need to first define the concept of a view.

Definition 1. *A view is a total order on a set of operations where each read to a variable returns the last value written to that variable.*

Intuitively, a view is how a single process saw the operations. Consistency models will be defined as sets of views that respect certain properties.

Definition 2. *An execution is a collection of views, with a one-to-one mapping from processes to views. In other words, each process has exactly one view.*

Definition 3. *If $<_R$ is a partial order on a set of operations O',*
then $respects(V, <_R, O')$ if V is a linear extension of $<_R$.

Now we need to define the orders that our consistency models will respect. First, there is the *process order* (also known as the program order), $<_{PO}$, which is the order in which the operations at each process happened in the program.

Definition 4. *The process order is $(o_1, i_1, x_1, n_1) <_{PO} (o_2, i_2, x_2, n_2)$ iff $i_1 = i_2 = i$ and $n_1 < n_2$.*

Orders that are process specific will be denoted with the process in brackets; for example, process i's process order is denoted $<_{PO[i]}$. Note that $<_P O[i]$ is a partition of $<_{PO}$, because each operation is issued by exactly one process, and, therefore, is in only that process's process order.

Now we can define sequential consistency.

Definition 5. *Sequential consistency is where all the processes observe the same view, V, and respects$(V, <_{PO}, (*, *, *, *))$.*

For causal consistency, we need the causal order, and, for that, we need the writes-to relation.

In order to know what value each read returns we need to know what write wrote the value. The writes-to relation is defined between these operations. We use $\perp$ for the initial value.

Definition 6. *The writes-to relation, $\mapsto_V$, for a view V over operations O' is*

$$\forall_{x \in V} \forall_{o_1 \in (r,*,x,*) \cap O'}$$
$$\quad \textit{if } \exists o_2 \in (w, *, x, *) \cap O' \textit{ s.t. } (o_2 < o_1 \textit{ in } V) \wedge$$
$$\quad\quad \neg \exists o_3 \in (w, *, x, *) \cap O' \textit{ s.t. } (o_2 < o_3 < o_1) \textit{ in } V$$
$$\quad\quad \textit{then, } o_2 \mapsto_V o_1$$
$$\quad \textit{otherwise } \perp \mapsto_V o_1$$

While it looks complicated, all definition 6 states is that $w \mapsto_V r$ if w was the last write to r's variable in the view. Then the $\mapsto$ for the execution is the union of the $\mapsto_V$ for the actual views of each process.

Definition 7. *The causal order for an execution is defined as follows: $o_1 <_{CO} o_2$ iff*

1. *$o_1 <_{PO} o_2$, or*
2. *$o_1 \mapsto o_2$, or*
3. *$\exists o_3$ such that $o_1 <_{CO} o_3 <_{CO} o_2$ (transitivity)*

Now we can define causal consistency.

Definition 8. *An execution is causally consistent if for the view, V_i, for each process, i, respects$(V_i, <_{CO}, (*, i, *, *) \cup (w, *, *, *))$.*

Basically causal consistency allows each process to have a largely independent view, but the view has to be consistent with the causal order. Furthermore, one process's view does not need to include another process's reads, which is why it is only defined over $(*, i, *, *) \cup (w, *, *, *)$.

We will notate combining two or more orders as below.

Definition 9. *$<_C = <_A \cup <_B$ means that $x <_C y$ if*

1. *$x <_A y$, or*
2. *$x <_B y$, or*
3. *$\exists z$ such that $x <_C z <_C y$ (transitivity)*

Illustrating Examples

In this section, executions will be represented as lists of operations for multiple processes. We will denote an operation by a specific process with a superscript, such as $W_X^p(v)$ for a write issued by process p, if it is not otherwise clear what process issued the operation.

Figures 1.1 and 1.2 show executions that illustrate the difference between sequential consistency and causal consistency. In figure 1.1, the following total order is legal and a linear extension of the process order: $W_X^1(1) < R_X^1(1) < W_X^2(2) < R_X^2(2) < R_X^1(2) < R_X^2(2)$. However, there is no single view for all processes for the execution shown in figure 1.2. Process order for process 2 indicates that $W_X^2(2) < R_X^2(2) < R_X^2(1)$. However, since $W_X^1(1)$ writes to $R_X^2(1)$, it must be in between $R_X^2(2)$ and $R_X^2(1)$, resulting in the order $W_X^2(2) < R_X^2(2) < W_X^1(1) < R_X^2(1)$. However, the same logic applied to process 1 results in the order $W_X^1(1) < R_X^1(1) < W_X^2(2) < R_X^1(2)$. In summary, a total order would have to have both $W_X^1(1) < W_X^2(2)$ and $W_X^2(2) < W_X^1(1)$, a contradiction.

On the other hand, causal consistency allows each process to observe writes in a different order, as long as its own reads read from the latest write in its order. Therefore, process 1 may view the order $W_X^1(1) < R_X^1(1) < W_X^2(2) < R_X^1(2)$, while process 2 sees $W_X^2(2) < R_X^2(2) < W_X^1(1) < R_X^2(1)$.

In both figures 1.1 and 1.2, the writes were not causally related, so they could be ordered in either way. However, if two writes are causally ordered, they must appear in that order in every process's view. An example of this is

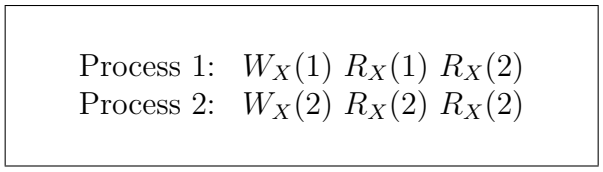


Figure 1.1: An execution that is sequentially consistent.

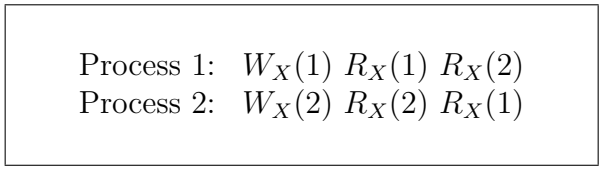


Figure 1.2: An execution that is causally consistent, but not sequentially consistent.

shown in figure 1.3. In this execution, in the causal order, $W_X(1) < W_Y(1)$ due to program order, and $W_Y(1) < R_Y(1)$ because $W_Y(1) \mapsto R_Y(1)$. Then $R_Y(1) < W_X(2)$ due to program order. Thus, by transitivity, $W_X(1) < W_X(2)$ in the causal order. This means that they must be in that order in all processes' views. Note that process 3's view must include $W_X(1) < R_X(1) < W_X(2) < R_X(2)$, which is, as we just showed, consistent with the causal order, which is shown in figure 1.4. In this figure, the partial order is indicated using arrows (i.e. $o_1 < o_2$ is equivalent to $o_1 \rightarrow o_2$).

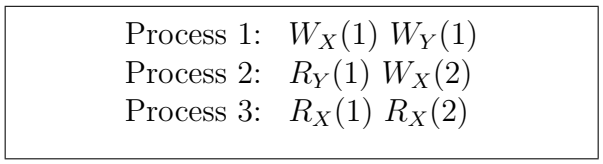


Figure 1.3: An execution that demonstrates causality.

On the other hand, figure 1.5 shows an execution that is not causal, because as before, in the causal order, $W_X(1) < W_X(2)$, but process 3's view must include $W_X(2) < R_X(2) < W_X(1) < R_X(1)$, which is not an extension of the causal order.

1.1.4 Record

In this work, we will consider records that consist of the order of pairs of operations (either at each process or globally). For example, one entry in the record might be "process 1's operation 21 happens before process 5's

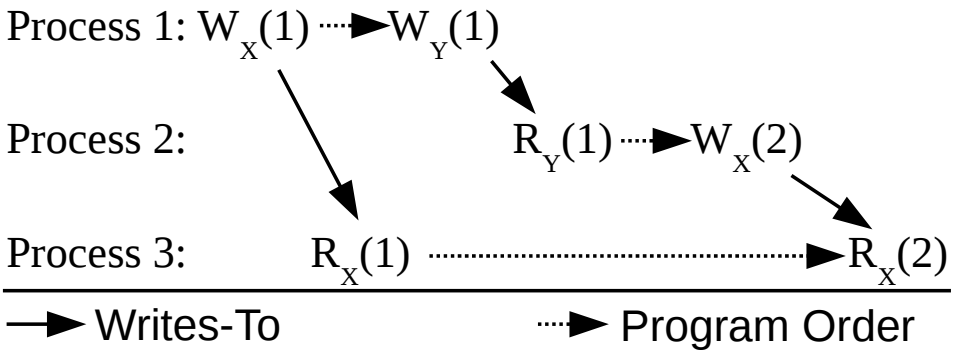


Figure 1.4: The causal order for the execution shown in figure 1.3.

Process 1:	$W_X(1)$ $W_Y(1)$
Process 2:	$R_Y(1)$ $W_X(2)$
Process 3:	$R_X(2)$ $R_X(1)$

Figure 1.5: An execution that is not causally consistent.

operation 8 in process 3's view." This sounds like a lot of information for each entry, but really, in process 3's record, it is just a pair $(p1o21, p5o8)$, where $p1o21$ and $p5o8$ are unique identifiers for process 1's operation 21 and process 5's operation 8, respectively.

This is sufficient for any consistency model where reads return the last value written, because the view can be specified fully by specifying the entire view through pairs of adjacent operations.

However, recording every pair is not necessary. The value that a read returns is completely determined by which write to that variable was the last before that read (in other words, which write wrote to the read). Thus, it is unnecessary to store the order between operations to different variables, because as long as the ordering between operations on the same variables matches the original execution, the values of the reads will also match the original execution. Therefore, we will define the data race order (DRO) as the total order on each variable (either at each process or globally). See figure 2.1 for an example of DRO. This gives us definitions 10 and 11.

Definition 10. *The data-race order for an execution with view V_i for process i, $<_{DRO[i]}$, is a relation such that* $\forall_{x \in X} \forall_{o_1, o_2 \in (w,*,x,*) \cup (*,i,x,*)} o_1 <_{DRO[i]}$

o_2 *iff* $o_1 < o_2$ *in* V_i.

Definition 11. *A record for process i,* $(<_{R[i]})$*, is a subset of* $(<_{DRO[i]})$*.*

There are a few other ways to create a record for RnR. Following immediately from definition would be recording the value of every read, then returning the recorded values for reads during re-execution. This requires storing information for every read, whereas storing the order may require much less when the consistency model guarantees certain orderings are impossible. Another way is to record the order of the underlying messages that implement the shared memory (if the shared memory is implemented on top of a message passing system). However, this would also store unnecessary information, as several messages may be sent for each operation. A fourth way is to split the execution into epochs during which there are guaranteed to be no races, and store what operations go into each epoch for each process. This is considered the state of the art for multiprocessors with a unified shared memory [2]. For example, in an epoch where there is no communication between processors, the new operations a process sees are only its own, which are ordered by the program itself. This largely relies on the fact that most variables are not shared, and that there is a coherence protocol (which maintains cache consistency) that does not generate communication while a single process is accessing a variable. However, several consistency models used in distributed systems (such as causal consistency) do not assume that the memories are coherent (the consistency model is not strictly stronger than cache consistency), and, thus, would not have a coherence protocol. This thesis, therefore, will focus on records that consist of ordering pairs of operations.

1.1.5 Replay

Definition 12. *A set of replay executions,* $\mathcal{V}$*, is data-race-correct iff*

$$\forall_{V \in \mathcal{V}} \forall_{i \in P} respects(V, <_{DRO[i]}, (w, *, *, *) \cup (*, i, *, *))$$

This is equivalent to the definition of correct replay given in [3]. Basically, what this means is that each process has to see the same view on a per variable basis in the replay as in the original execution. So, for each variable,

at each process, the exact same total order of reads and writes (including other process's writes) happens in the replay as the original execution. This is slightly stronger than is actually needed. For example, suppose a variable is written to 3 times, and then read once, after all 3 writes. Then, it does not actually matter in what order the first 2 writes happen; it only matters that the same write is last in the original and the replay. However, for definition 12 the first two writes have a specified order, and this is the definition for correct replay that will be used in this thesis.

CHAPTER 2

RELATED WORK

2.1 Causal Consistency

There are many relaxed consistency models. Many works investigate or use causal consistency [4], [5], [6], [1], [7]. No other work that we are aware of investigates RnR in causal consistency. Several works also have to adjust the definition of causal consistency slightly for their application [5], [6]. This thesis will use a modified definition of causal consistency as well, in order to make sure the executions are "replayable", which is explained in chapter 3.

2.2 Record and Replay

Several works [8], [9], [2] investigate how to do RnR on consistency models weaker than sequential consistency. However, they rely on maintaining a consistency condition called cache consistency (or, more commonly, in the case of multiprocessors, cache coherence). Several consistency models of interest to distributed computing such as causal consistency and eventual consistency do not maintain this condition.

2.3 Optimal Record and Replay

Several previous works [3], [8], [9], [2] deal with reducing the size of records for RnR. However, only one previous work [3], to our knowledge, has addressed the question of what the *minimum* record is. This work only does this in the context of sequential consistency.

In 1993, Robert Netzer created an algorithm to find the optimal record (of data race ordering) for RnR [3]. The result implicitly assumes that correct

replay means reproducing the data race ordering, or in other words, resolving every data race identically to the original execution. There are a few cases where this is unnecessary, which was discussed in 1.1.5.

The optimal solution is to take the union of the data race order and program order and find the covering relation of that union. Since the program order is available to both the original execution and the replay, the record consists of the covering relations that are not in the program order. Finding the covering relations for a finite set is equivalent to finding the transitive reduction of a directed acyclic graph.

Figure 2.1 shows an example execution with a partial order given by the union of DRO and process order. Note that most of the orderings in the data race order are covered by other orderings. For example, $W_X(1) < W_X(2)$ in DRO, but it is also true that $W_X(1) < W_Y(1) < W_X(2)$ in the program order, so the DRO provides no extra information in this case and would not be in the transitive reduction. Figure 2.2 shows the transitive reduction of the same partial order.

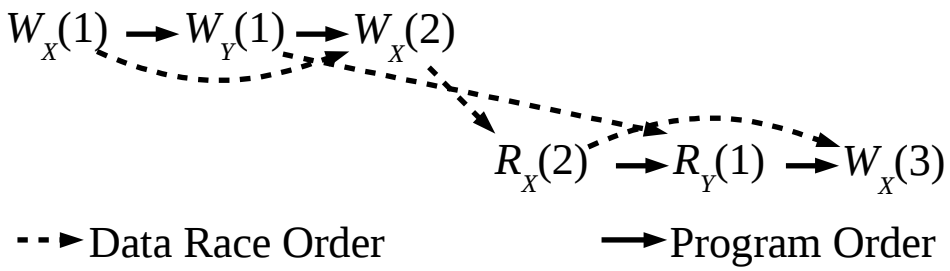


Figure 2.1: An execution on two processes (top and bottom) showing the program order and data race order.

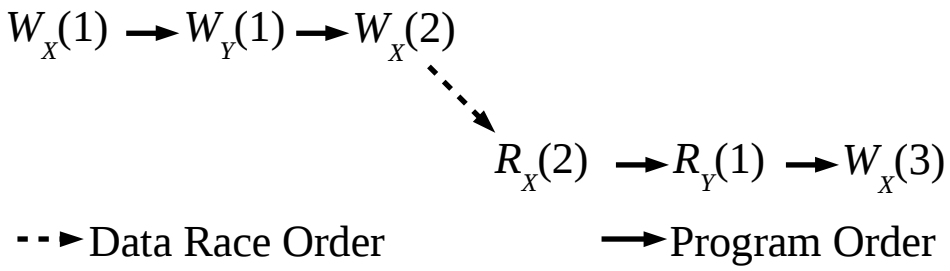


Figure 2.2: The transitive reduction of the execution in figure 2.1.

The best known general algorithm for transitive reduction (equivalent to Boolean matrix multiplication) requires $O(n^2)$ time [10]. However, due to the restricted nature of these orders, Netzer's algorithm can run in linear time [3].

CHAPTER 3

REPLAYABILITY

Consider the following method for replaying an execution: For each entry in the record, $a <_{R[i]} b$, one of two things will happen:

1. If i issued b, i will not issue b until it has performed a.

2. If b is a write issued by another process, it will not perform b until it has performed a, even if it receives b before a.

In this way, $a <_{R[i]} b$ is enforced in the replay, and if an operation does not occur in any record entry the replay system can go ahead with it and proceed without any additional work (other than what is necessary to maintain consistency).

However, there are some executions that are causally consistent but will not make progress in this replay system. For example, figure 3.1 shows an execution that is not replayable. It is causally consistent; for both processes, no writes are issued after any read, so the writes-to relation for each process's reads does not affect the other process's view. Thus, any view that is a linear extension of the program order is consistent. Figure 3.1b shows one such set of views. However, note that in this view, process 1 sees $W_X(2)$ before $W_X(1)$ and process 2 sees the opposite. Therefore, in the replay, process 1 will not issue $W_X(1)$ until $W_X(2)$ is issued by process 2, but process 2 will not issue $W_X(2)$ until $W_X(1)$ is issued by process 1; there is a circular dependence. In other words, neither process will issue any operation, resulting in a deadlock during replay.

This thesis assumes the above model of replay, and the RnR algorithm will only target causal executions that are replayable. A real system can easily eliminate unreplayable executions by following two rules:

- Any time a process issues a write, apply it locally before propagating it to other processes.

Process 1:	$W_X(1)$ $W_Y(1)$ $R_Y(2)$ $R_X(1)$
Process 2:	$W_X(2)$ $W_Y(2)$ $R_Y(1)$ $R_X(2)$

(a) The execution.

Process 1:	$W_X(2)$ $W_X(1)$ $W_Y(1)$ $W_Y(2)$ $R_Y(2)$ $R_X(1)$
Process 2:	$W_X(1)$ $W_X(2)$ $W_Y(2)$ $W_Y(1)$ $R_Y(1)$ $R_X(2)$

(b) One possible view for each process.

Figure 3.1: An execution that is not replayable.

- For any non-local write, do not order it before any write that you have already issued.

In other words, each process makes sure that it is the first process to see its own writes, which is a very natural implementation rule. We will formally define what it means for an execution to be replayable.

Definition 13. *A total order, $<_\pi$, over a set of operations, O_π, is a prefix of a view, $<_V$, if*

1. $a <_\pi b \implies a <_V b$, *and*

2. *if* $a <_V b$, *then* $(b \in O_\pi \implies a \in O_\pi)$

Basically, a prefix is just the beginning of a total order (where the length of the beginning can be anything). Let P be the number of processes.

Definition 14. *A causal execution prefix, $(<_{\pi[1]}, <_{\pi[2]}, ..., <_{\pi[P]})$, is one prefix for each process's view in a consistent execution (for whatever consistency model is being used in the system), such that if $\exists i$ such that $o \in O_{\pi[i]}$, then $o \in O_{\pi[j]}$, where j is the process that issued o.*

Basically, a causal execution prefix means that if any process sees an operation then the issuing process should have seen it. The reason we call it "causal" is because the cause of an operation is some process issuing that operation.

Definition 15. *An execution is "replayable", if for every causal execution prefix, $(<_{\pi[1]}, ..., <_{\pi[P]})$, $\exists p \in$ processes, $\exists o_1, o_2 \in$ operations, such that o_1*

is that last operation in $<_{\pi[p]}$ *and* $o_2 \notin O_{\pi[p]}$ *and* $(<_{\pi[1]}, ..., <_{\pi[p]} \cup \{o_1 < o_2\}, ..., <_{\pi[P]})$ *is a causal execution prefix.*

Basically, this means that if the program state is equivalent to any causal execution prefix, the system can make progress without becoming a non-causal execution prefix. Making progress is equivalent to issuing new operations or seeing new operations from other processes; the new operation is o_2 in the definition.

Consider the example in figure 3.1. As discussed above, it is not replayable. This can be derived from definition 15; the empty prefix (which is a causal execution prefix) has no way of adding a single operation and staying a consistent prefix. If $W_X(2)$ is added to process 1's view, then it is not a causal execution prefix because $W_X(2)$ has not happened at the issuing process. If $W_X(1)$ is added to process 2's view, then it is not a causal execution prefix because $W_X(1)$ has not happened at the issuing process. Note that every causally consistent view begins with $W_X(2)$ at process 1 or $W_X(1)$ at process 2, so if any other operation is added, then the resulting prefix is not causally consistent.

CHAPTER 4

CAUSAL CONSISTENCY

This chapter will show that an algorithm similar to Netzer's algorithm can produce the minimum record for causal consistency. Netzer's algorithm is described in section 2.3. For causal consistency each process works independently to build a record.

Only one function needs to be added to a causally consistent distributed memory system to generate the record. This function, like Netzer's original algorithm, finds the transitive reduction of DRO and causal order. However, for causal consistency, this order is a per process order rather than a global order.

Before presenting the modified algorithm, we will introduce some notation, and prove that finding the transitive reduction of DRO and causal order is sufficient and necessary for causal consistency just like it is for sequential consistency. The proof is somewhat more complex for causal consistency, because the causal order depends heavily on the writes-to relation, which is dependent on what view actually occurs in the replay execution.

4.1 Decomposed Causal Consistency

The causal order can be decomposed into two orders: Program order and Write-Read-Write order (defined below) [1]. In other words, respecting $<_{PO} \cup <_{WO}$ is sufficient to enforce causal consistency. Also, $\mapsto$ must be respected, but the definition of view (definition 1) ensures this.

Definition 16. *Two writes are ordered by write-read-write order,* $w_1 <_{WO} w_2$*, iff there exists a read,* r*, such that* $w_1 \mapsto r <_{PO} w_2$*.*

Write-read-write order, $<_{WO}$, captures the same thing that the writes-to, $\mapsto$, captures in the earlier definition of causal consistency, for two reasons.

First, since reads are not seen by processes other than the issuing process, it does not matter that the reads are not ordered, since the causal order is the global order; reads still must be after the write that writes to them, based on definition 1. Second, all the other operations ordered by the $\mapsto$ ordering are still ordered.

Write-read-write order, $<_{WO}$, captures the same thing that the writes-to, $\mapsto$, captures in the earlier definition of causal consistency, for two reasons. First, since reads are not seen by processes other than the issuing process, it doesn't matter that the reads are not ordered, since the causal order is the global order; reads still must be after the write that writes to them, based on definition 1. Second, all the other operations ordered by the $\mapsto$ ordering are still ordered.

4.2 Minimum Record for Causal Consistency

4.2.1 Overview

In this section we will derive the minimum record for RnR in a causally consistent system. Section 4.2.2 restricts the executions to replayable executions. The minimum record will only be derived for these executions. Section 4.2.3 discusses precisely what the minimum record is, while 4.2.4 proves that this is, in fact, the minimum record. After the proof, section 4.3 describes an algorithm that records this minimum record. Finally, section 4.4 shows (by counter-example) that eliminating the unreplayable executions was necessary for the proof, as what we prove about replayable executions does not hold for unreplayable ones.

4.2.2 Replayable Causal Consistency

The record only needs to handle replayable executions (it would be useless for an unreplayable execution), so the proof will be limited to replayable executions. Unreplayable executions are caused by circular dependence. In order to prevent circular dependence, the write-read-write order is replaced with strong write order.

Definition 17. *Strong write order is: If* $a <_{DRO[process(b)]} b <_{PO} c$*, then* $a <_{SWO} c$*.*

Now we define a consistency model that is slightly stronger than causal consistency, called *replayable causal consistency*. Since $<_{CO}=<_{PO} \cup <_{WO}$, it is clear that the only difference between causal consistency and replayable causal consistency is that $<_{WO}$ in causal consistency is replaced with $<_{SWO}$ in replayable causal consistency (cf. definition 8).

Definition 18. *Replayable causal consistency is where for each process* i*, for its view in the execution,* V_i*,* $respects(V_i, <_{PO} \cup <_{SWO}, (*, i, *, *) \cup (w, *, *, *))$*.*

Lemma 1. $a <_{WO} c \implies a <_{SWO} c$

Proof. Consider any relation $a <_{WO} c$. By definition, $\exists b$, such that $a \mapsto b <_{PO} c$. Then, by definition of $\mapsto$, $a <_{DRO[process(b)]} b <_{PO} c$. Thus, $a <_{SWO} c$. □

Due to lemma 1, if the replay recreates the strong write order, it also recreates the write-read-write order. The strong write order still needs to be inferred from the record, because $a <_{DRO[process(b)]} b$ may not be in the record and may need to be inferred.

4.2.3 The Minimum Record

Notation: For any order $<_A$, $\lessdot_A$ denotes the covering relation of that order. In other words, it is all the relations in the transitive reduction of $<_A$.

Definition 19.

$$<_{DRO^*[i]}=$$

$$\{ \quad a < b : \\ a \lessdot b \ in \ <_{PO} \cup <_{SWO} \cup <_{DRO[i]} \\ \quad and \ a \not<_{PO} b \\ \quad and \ a \not<_{SWO} b \qquad \}$$

Furthermore, $<_{DRO^*}$ *refers to the per process collection of all of* $<_{DRO*[i]}$*.*

In section 4.2.4, it will be proven that $<_{DRO^*}$ is the minimum record. Note that any relation in the transitive reduction must be in one of $<_{PO}$, $<_{SWO}$, or $<_{DRO[i]}$, and the record is a subset of $<_{DRO[i]}$, which is the definition of a proper record. Furthermore, this definition eliminates those relations that happen to be $<_{PO}$ or $<_{SWO}$ in addition to $<_{DRO[i]}$. A transitive reduction also, by definition, removes those relations that are covered by a longer transitive path in the partial order.

We will use the notation $<_A + <_{DRO}$ to indicate a collection, on a per process basis of $<_A \cup <_{DRO[i]}$. That is, for process i, $<_A + <_{DRO}$ is $<_A \cup <_{DRO[i]}$, and for process j, it is $<_A \cup <_{DRO[j]}$.

In the replay, we only directly enforce $<_{PO} + <_{DRO^*}$, so it is a priori unknown whether the same $<_{SWO}$ is enforced during replay. We do enforce replayable causal consistency, so some $<_{SWO}$ will exist in the replay. First, we know that if $a <_{DRO*[process(b)]} b <_{PO} c$, then $a <_{SWO} c$ in the replay. However, for $a <_{DRO[process(b)]} b$ that are not in the record, we will have to show that $a < b$ in b's view in every possible replay execution. We will define $<_{SWO^*}$ as the relations in $<_{SWO}$, that we can infer from the record.

Definition 20. *$a <_{SWO^*} b$ if and only if for all replayable causally consistent executions that are consistent with $<_{PO} + <_{DRO^*}$, $a <_{SWO} b$.*

Thus, the replay will be consistent with $<_{PO} \cup <_{SWO^*} + <_{DRO^*}$. To show that the replay is correct, it is necessary to show $<_{PO} \cup <_{SWO^*} + <_{DRO^*}$ is equivalent to $<_{PO} \cup <_{SWO} + <_{DRO}$ (which is theorem 1).

4.2.4 Proof

We define a single relation, the replayable causal relation, for verification that there is no circular dependence. Intuitively, the difference between causal order and replayable causal relation is that, in the causal order, a write has to be read by a process before it is causally ordered for that process, but in the replayable causal order a write is causally ordered for a process if any operation to that variable is performed at that process.

Definition 21. *The replayable causal relation, $<_{RCO}$, is: $a <_{RCO} b$ if*

1. *$a <_{PO} b$, or*

2. $a <_{DRO[process(b)]} b$, *or*

3. $\exists c$ *such that* $a <_{RCO} c <_{RCO} b$ *(transitivity), or*

4. $\exists c$ *such that* $a <_{process(b)]} c <_{RCO} b$

Conjecture 1. *An execution is replayable iff the replayable causal relation is a partial order (that is, it is asymmetric).*

The above conjecture would connect definition 21 to the definition of *replayable* in chapter 3. In this section, all the theorems assume $<_{RCO}$ is a partial order. A case where $<_{RCO}$ is not a partial order is addressed in section 4.4.

First, we prove a short lemma connecting the replayable causal relation to the replayable causal consistency ($<_{PO} \cup <_{SWO}$). This will allow us to use replayability (based on the $<_{RCO}$) in the main proof, while the consistency model is only restricted to replayable causal consistency ($<_{PO} \cup <_{SWO}$).

Lemma 2. *If* $a < b$ *in* $<_{PO} \cup <_{SWO} \cup <_{DRO[process(b)]}$, *then* $a <_{RCO} b$.

Proof. This follows trivially from the definition of $<_{RCO}$: $<_{PO}$ is included in $<_{RCO}$ due to rule 1; $<_{DRO[process(b)]}$ is included in $<_{RCO}$ due to rules 2 and 4; and $<_{SWO}$ is, by definition, the application of rule 2 followed by rule 1.

□

Theorem 1 is the main proof for the minimum record. It shows that all the $<_{SWO}$ can be inferred given the minimum record and replayable causal consistency (that is, there are no consistent replay executions where $<_{SWO}$ is not the same as the original execution).

Theorem 1. $<_{SWO^*}=<_{SWO}$ *if* $<_{RCO}$ *is a partial order for the original execution.*

Proof. By induction.

Inductive Hypothesis: $\forall o_n <_{RCO} o_N$, if $o_k \lessdot_{SWO} o_n$ then $o_k \lessdot_{SWO^*} o_n$.

Base Case: $\neg\exists o_n <_{RCO} o_N$**.** Then, by lemma 2, $\neg\exists o_k$ such that $o_k <_{SWO} o_N$. The claim, $o_k \lessdot_{SWO} o_N \implies o_k \lessdot_{SWO^*} o_N$, is vacuously true.

Induction

Case 1: $\neg\exists o_k$ **such that** $o_k <_{SWO} o_N$**.** The claim, $o_k \lessdot_{SWO} o_N \implies o_k \lessdot_{SWO^*} o_N$, is vacuously true.

Case 2: $\exists o_k$ **such that** $o_k <_{SWO} o_N$**.** By definition of $<_{SWO}$, $\exists o_j, i$ such that o_j is an operation at process i, and $o_k <_{DRO[i]} o_j <_{PO} o_N$.

- Case 2.1: $o_k <_{DRO^*[i]} o_j$. Then, we can infer directly that $o_k <_{SWO^*} o_N$.
- Case 2.2: $o_k \not<_{DRO^*[i]} o_j$. By the definition of transitive reduction, there exists some path in the transitive reduction of $<_{DRO[i]} \cup <_{PO} \cup <_{SWO}$ from o_k to o_j. All the edges in this path are $\lessdot_{DRO^*[i]}$ or $<_{PO}$ or $<_{SWO}$. Note that all the edges in $\lessdot_{DRO^*[i]}$ or $<_{PO}$ are already in the replay, and it is sufficient to show that the $\lessdot_{SWO}$ can be inferred. Consider any edge $a \lessdot_{SWO} b$ in this path. Note that $b < o_N$ in $<_{DRO[process(o_N)]} \cup <_{PO} \cup <_{SWO}$, so, by lemma 2, $b <_{RCO} o_N$. So, by the inductive hypothesis, $a \lessdot_{SWO^*} b$. We can infer $o_k <_{DRO[i]} o_j$, which implies $o_k <_{SWO^*} o_N$.

□

Theorem 2. *The minimum record for replayable causal consistency is* $<_{DRO^*}$ *if* $<_{RCO}$ *is a partial order for the original execution.*

Proof. Theorem 1 shows that the record is sufficient.

Note that no $\lessdot_{DRO^*[i]}$ relation can be inferred from any $\lessdot_{DRO^*[j]}$, since, by definition of replayable causal consistency, the views are not dependent on each other except through the $<_{PO} \cup <_{SWO}$. Furthermore, they cannot be inferred from $<_{PO} \cup <_{SWO}$, because those that were implied by that were explicitly removed in the transitive reduction or the "$a \not<_{PO} b$ and $a \not<_{SWO} b$" part of the definition of $\lessdot_{DRO^*[i]}$.

In any partial order, if two elements are not ordered, there exist linear extensions of the partial order where they appear in either order [11]. If you remove any $a <_{DRO^*} b$ from any record that is a subset of $<_{DRO}$, then (since it is in the transitive reduction and not inferrable) a and b are no longer ordered. Thus, there exists a view where b occurs before a. This view is only in incorrect replay executions.

□

Note that the above proof is for the *minimum* record, not just the *minimal* record, since it proves that all the record entries are necessary in *any* record, not just records that have removed the $<_{DRO}$ relations that $<_{DRO^*}$ does.

4.3 Algorithm

This section describes a real time algorithm to find the minimum record for RnR in a causally consistent system.

4.3.1 Vector Functions and Comparison used in Pseudocode

The comparison $\leq$ for vectors (of equal length) in the pseudocode will mean the following: $A \leq B$ if every element in A is less than or equal to the corresponding element in B. Also, $A < B$ if $A \leq B$ and $A \neq B$. Furthermore, $(A \leq B \iff B \geq A)$ and $(A < B \iff B > A)$.

The function $Max(A, B)$ on two vectors returns the vector Y such that $Y_i = Max(A_i, B_i)$ for all i in $\{1, 2, ..., \text{vector width}\}$. In other words it is the element-wise max and it is possible that $Y \neq A$ and $Y \neq B$.

The vector $\hat{\imath}$ refers to the vector with 1 in entry i and 0 in all other entries.

4.3.2 Pseudocode

The following pseudocode is running on process i, and receives an operation that was issued by process j (and it may be that $i = j$).

On Operation (op, j, X, n):

1: **if** $i = j$ **then**
2: `initial_vector` := `last_operation.vector` $+ \hat{\imath}$
3: **else**
4: `initial_vector` := (op, j, X, n).`vector`
5: **end if**
6: **if** `initial_vector` $\not\geq$ `last_event`$[X]$.`vector` **then**
7: $RecordRace$((`last_event`$[X]$, (op, j, X, n)))
8: **end if**
9: (op, j, X, n).`vector` := Max(`initial_vector`, `last_event`$[X]$.`vector`)
10: `last_event`$[X]$:= (op, j, X, n)
11: **if** $i = j$ **then**
12: `last_operation` := (op, j, X, n)
13: **end if**

Also, on writes, (w, i, X, n).`vector` is sent to the other processes with the

write, which is read by the other processes in line 4. This is why line 4 can read the operations vector before it is set (line 9); the vector in line 4 is the vector according to the process that issued the write (process j), whereas the vector in line 9 is the vector according to the current process (process i).

4.3.3 Correctness

Lemma 3. *For process i, $\forall o_1, o_2 \in$ operations, $o_1.\texttt{vector} < o_2.\texttt{vector}$ iff $o_1 < o_2$ in $<_{DRO[i]} \cup <_{PO} \cup <_{SWO}$.*

Proof. Every relation in $\lessdot_{PO}$ makes the vector on the right larger due to line 2. Every relation in $\lessdot_{DRO[i]}$ makes the vector on the right larger due to line 9. Every relation in $o_1 \lessdot_{SWO} o_2$ is due to $o_1 \lessdot_{DRO[process(o_3)]} o_3 <_{PO} o_2$. Due to line 9, $o_1.\texttt{vector} < o_3.\texttt{vector}$, and $o_3.\texttt{vector} < o_2.\texttt{vector}$ due to line 2. Thus, if $<_{DRO[i]} \cup <_{PO} \cup <_{SWO}$, then $o_1.\texttt{vector} < o_2.\texttt{vector}$.

Furthermore, line 9 and 2 are the only lines that can increase the vector, so $o_1.\texttt{vector} < o_2.\texttt{vector}$ only if they are in the transitive closure of $<_{PO}$ and $<_{DRO[j]}$ where $j = i$, or some process that has written a value that i has seen. This is precisely what $<_{DRO[i]} \cup <_{PO} \cup <_{SWO}$ is, so if $o_1.\texttt{vector} < o_2.\texttt{vector}$, then $<_{DRO[i]} \cup <_{PO} \cup <_{SWO}$. □

Theorem 3. *The algorithm described above records the minimum record.*

Proof. The algorithm only records a race if $\texttt{initial\_vector} \not\geq \texttt{last\_event}[X].\texttt{vector}$. However, `initial_vector` is just that operation's vector before the process has adjusted due to $\lessdot_{DRO[i]}$. Thus, if `initial_vector` is greater than the previous operation's vector, then, by lemma 3, if we did not include this relation, these operations are unordered. Therefore, this relation must be in the transitive reduction of $<_{DRO[i]} \cup <_{PO} \cup <_{SWO}$ (as transitive reduction preserves order). Also by lemma 3, the converse is true. By theorem 2, this is the minimum record. □

4.4 Minimum Record for Unreplayable Executions

The proof showed that the transitive reduction of $<_{DRO[i]} \cup <_{PO} \cup <_{SWO}$ is sufficient for executions that are replayable (in the sense that $<_{RCO}$ is a

partial order). This section will show that the same transitive reduction is not sufficient for some executions that are not replayable.

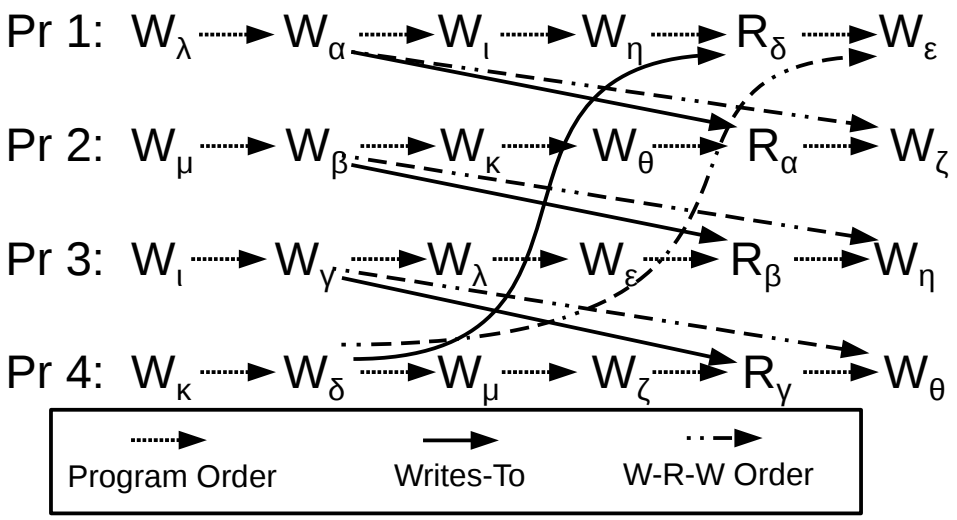


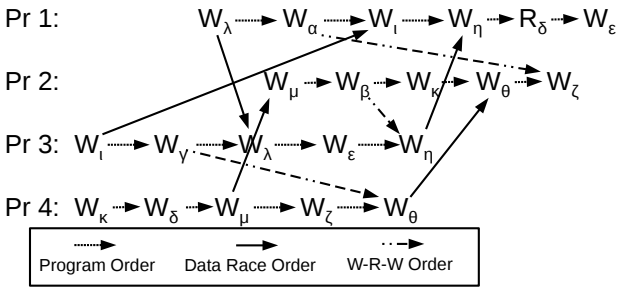
Figure 4.1: The writes-to and causal order for the unreplayable execution.

Consider the execution shown in figure 4.1. In this example we omit the values of the operations, since they are largely inconsequential; instead the writes-to order indicates which write wrote to which read. As in the previous examples, the variable for the operation is shown in the subscript. In each row are the operations that that process issued.

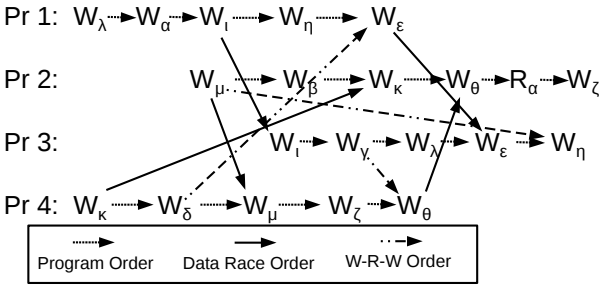
One possible original execution that could produce this execution is shown in figure 4.2. Note that for none of the processes is the $\lessdot_{DRO}$ edge that enforces the writes-to relation (i.e. $W_\delta <_{DRO[1]} R_\delta$) in the transitive reduction. In all cases, the path that allows this edge to not be in the transitive reduction is one of the $\lessdot_{WO}$ edges.

The execution is not replayable; the replayable causal relation is not a partial order, as is shown in figure 4.3. It is not a partial order as there are cycles, for example, from process 1's W_ϵ to process 3's W_ϵ and back. These cycles exist because each process sees its own final two writes after the final two writes of the other process writing to the same variables. For example, process 1 sees process 3's W_ϵ before its own, whereas process 3 sees process 1's W_ϵ before its own.

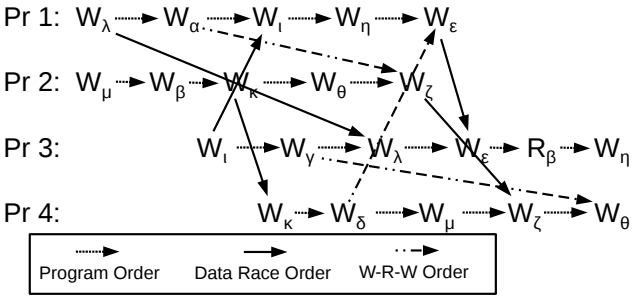
Of course, $\lessdot_{WO}$ edges are not included in the record, instead they must



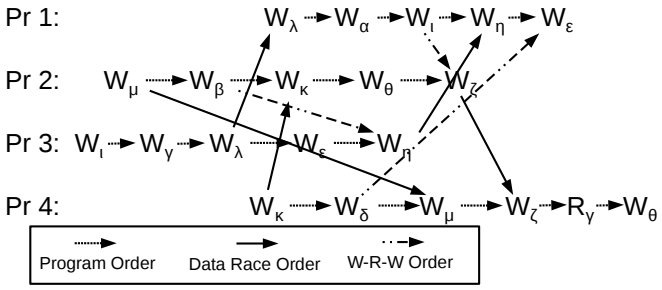
(a) Process 1's view (transitive reduction).



(b) Process 2's view (transitive reduction).



(c) Process 3's view (transitive reduction).



(d) Process 4's view (transitive reduction).

Figure 4.2: The views for the unreplayable execution, including the causal order.

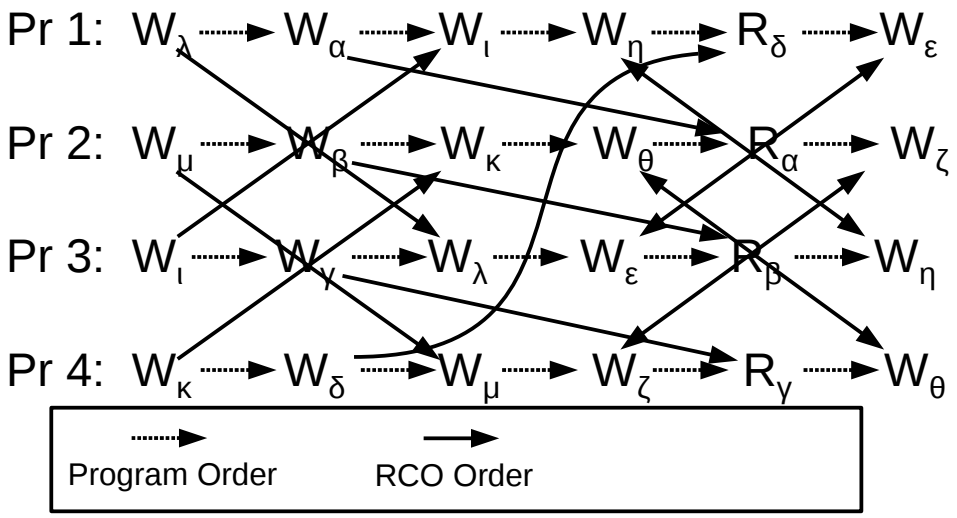
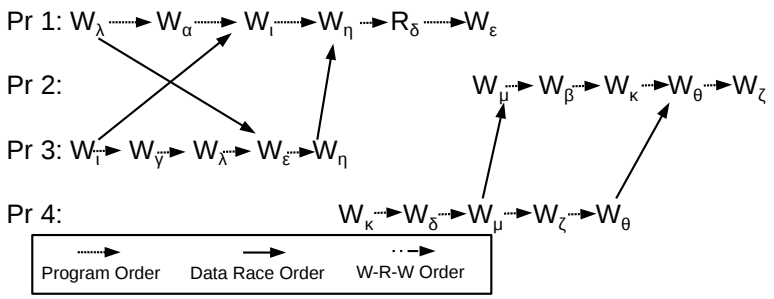
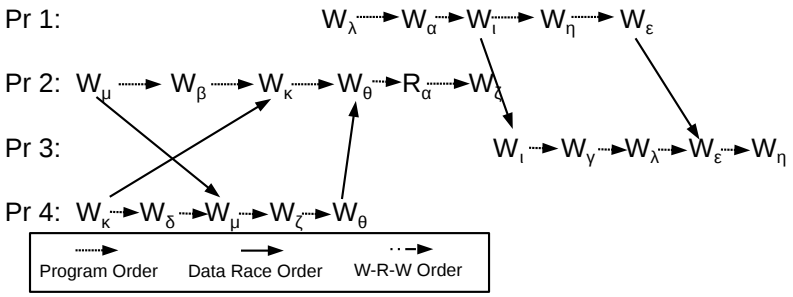


Figure 4.3: The replayable causal relation for the unreplayable execution.

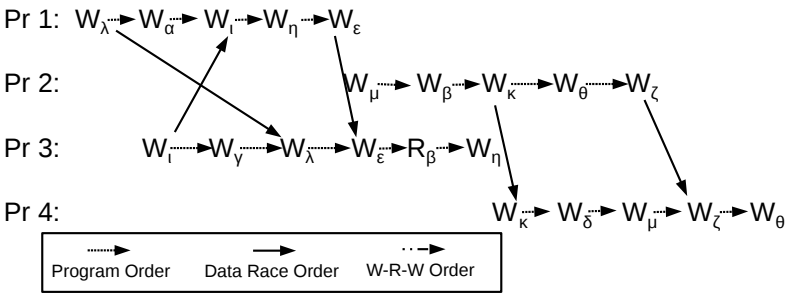
be inferred. If you start with just the record (from the transitive reduction) and the program order, the writes-to relations that they are dependent on cannot be inferred. It is possible that the writes happen after the read, as shown in figure 4.4. We do not assume any specific model of replay here, just that the replay execution must be causally consistent and each process's view must be a linear extension of the record. This execution is consistent with the record and causally consistent, but it is not a correct replay. As proven in section 4.2.4, this could not happen with a replayable execution. This shows that some constraint had to be placed on causal consistency in order for the above proof to be correct.



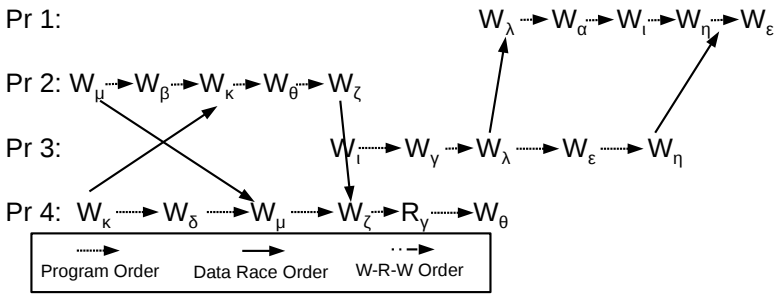
(a) Process 1's view (transitive reduction).



(b) Process 2's view (transitive reduction).



(c) Process 3's view (transitive reduction).



(d) Process 4's view (transitive reduction).

Figure 4.4: A possible "replay execution" for the unreplayable execution that is consistent with the record, but not the original execution.

CHAPTER 5

HETEROGENEOUS CONSISTENCY RECORD AND REPLAY

Consider the following situation: You have recorded an execution on a causally consistent system, but the system that you are doing the replay on only guarantees local consistency (defined below). We will call recording on a system with one consistency model and replaying on a system with another consistency model *heterogeneous consistency record and replay.*

Definition 22. *An execution is locally consistent iff for each process i, there exists a view, V_i, of the execution such that respects$(V_i, <_{PO[i]}, (*, i, *, *) \cup (w, *, *, *))$.*

In other words, the processes only have to respect their own program order, and can see other processes writes in *any* order. Consider the execution shown in figure 5.1. In this figure, the super scripts are there just to give each read a unique identifier. The minimum record under causal consistency for process 2 is $W_X(2) <_{RCO[2]} R_X^\alpha$. However, this is insufficient to reproduce the execution under a system that is only locally consistent, because $W_X(2) < R_X^\alpha(2) < W_X(1) < R_X^\beta(1)$ is a valid view under local consistency (because, under local consistency, one process does not even have to observe another process's program order, only its own). This view is also consistent with the record, but it is clearly different from the original execution.

It is unsurprising that you need more information to reproduce an execution under local consistency than causal consistency, because the causal consistency, being a stronger rule, itself contains information about which executions are legal. However, the surprising result for causal consistency (shown in the following section) is that replay does not actually require more information than sequential consistency. Causal consistency is strictly weaker than sequential consistency, just as local consistency is strictly weaker than causal consistency, but does not require any additional record entries.

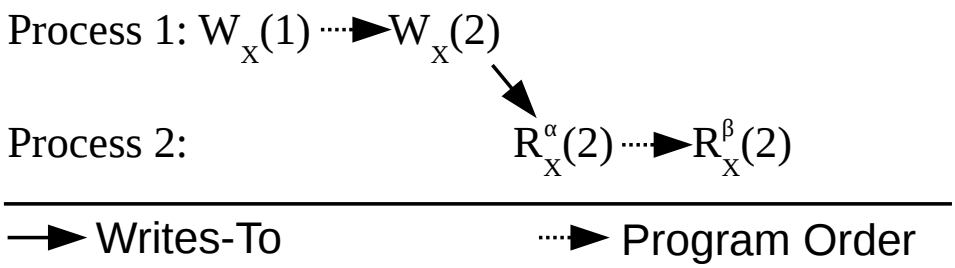


Figure 5.1: An execution that is causally consistent (and also process consistent).

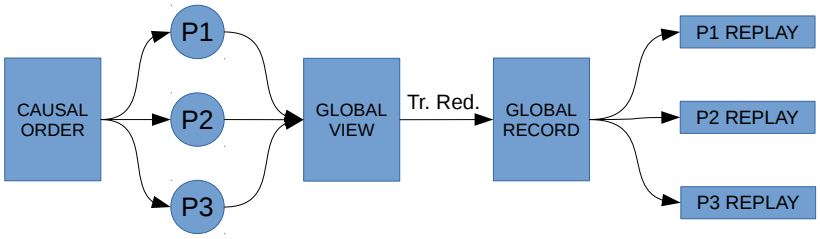
5.1 Replaying Sequentially Consistent Executions

Note that the minimum record for causal consistency is just the transitive reduction of the data-race order and the causal order. Interestingly, the minimum record for sequential consistency is the same thing (as shown by [3]); the difference is that for sequential sequential consistency, the data-race order is global, whereas for causal consistency, the data-race order is per process. This is shown in figure 5.2.

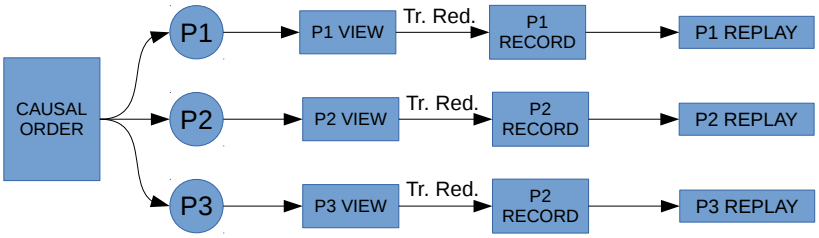
Note that if a sequentially consistent execution ran on our algorithm, rather than Netzer's, it would have the same result, because the only difference is that our algorithm runs on different processes' views (which, by the definition of sequential consistency, are identical) instead of a global view. Furthermore, sequentially consistent executions are replayable causally consistent, because sequential consistency is strictly stronger than replayable causal consistency. Therefore, the replay on the causally consistent system must be correct, as proven in chapter 4. Thus, as shown in figure 5.3, the minimum record for sequential consistency will allow replay in causal consistency.

5.2 Gains From Heterogeneous Consistency Replay

Heterogeneous consistency replay could reduce the cost of replay because causal consistency can be implemented with less communication cost than sequential consistency. For example, in a sequentially consistent system, a process may need to make sure that no other process is writing concurrently before proceeding with a write; however, the same write in causal consistency



(a) Netzer's algorithm for the minimum record for sequential consistency.



(b) Our algorithm for the minimum record for causal consistency.

Figure 5.2: Comparison of the algorithms for minimum record for sequential and causal consistency.

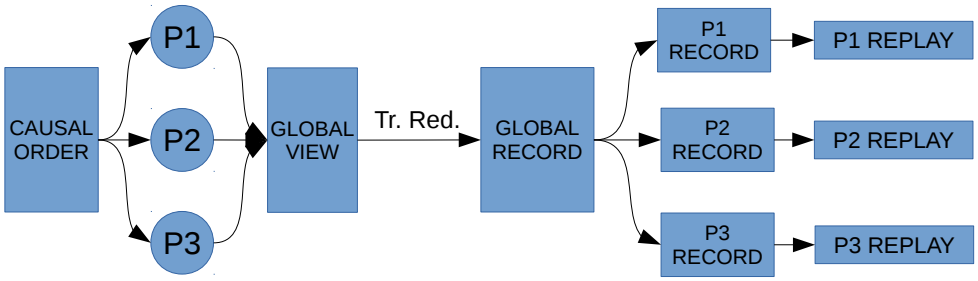


Figure 5.3: Heterogeneous consistency replay from sequential to causal consistency.

can proceed immediately (unless it is waiting for a write that is in the record). The magnitude of these benefits is unknown, and would depend both on application and implementation. The fact that correct replay under causal consistency is effectively free shows that this is a promising area of research.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Each of the contributions in this paper was only one example of a new research topic that could be more fully explored.

We showed that the replay mechanism can depend on the consistency model. Future work should investigate what replay mechanisms should be used.

We showed that minimal record for RnR depends on the consistency model. In particular, we developed an algorithm to find the minimum record for causal consistency. However, there is still a vast space of consistency models for which the minimum record is unknown.

We showed that one can replay executions from one consistency model on a system that implements a different consistency model. In particular, we showed that the minimum record for sequential consistency is sufficient to replay a sequentially consistent execution on a causally consistent system. Future work could generalize this: Is there a general way to know how much information you need to replay an execution on a different consistency model?

In conclusion, RnR for distributed shared memory is a deep topic. As demonstrated, it can be easily optimized, there are multiple ways to do it, and it can vastly help distributed system programmers.

REFERENCES

[1] R. C. Steinke and G. J. Nutt, "A unified theory of shared memory consistency," *J. ACM*, vol. 51, no. 5, pp. 800–849, Sep. 2004. [Online]. Available: http://doi.acm.org/10.1145/1017460.1017464

[2] N. Honarmand and J. Torrellas, "Relaxreplay: Record and replay for relaxed-consistency multiprocessors," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 223–238, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2654822.2541979

[3] R. H. B. Netzer, "Optimal tracing and replay for debugging shared-memory parallel programs," *SIGPLAN Not.*, vol. 28, no. 12, pp. 1–11, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/174267.174268

[4] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49. [Online]. Available: http://dx.doi.org/10.1007/BF01784241

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465279 pp. 761–772.

[6] M. Perrin, A. Mostefaoui, and C. Jard, "Causal consistency: Beyond memory," 2016. [Online]. Available: http://arxiv.org/abs/1603.04199

[7] R. Friedman, R. Vitenberg, and G. Chockler, "On the composability of consistency conditions," *Information Processing Letters*, vol. 86, no. 4, pp. 169 – 176, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019002004982

[8] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Respec: Efficient online multiprocessor replayvia speculation and external determinism," *SIGPLAN Not.*, vol. 45, no. 3, pp. 77–90, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1735971.1736031

[9] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346273 pp. 121–130.

[10] P. E. O'Neil and E. J. O'Neil, "A fast expected time algorithm for boolean matrix multiplication and transitive closure," *Information and Control*, vol. 22, no. 2, pp. 132 – 138, 1973. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019995873902283

[11] W. T. Trotter, *Combinatorics and Partially Ordered Sets: Dimension Theory*, ser. John Hopkins Series in the Mathematical Sciences. Baltimore and London: The John Hopkins University Press, 1992.