UNIVERSITY OF CALIFORNIA
Los Angeles

Prove Once, Run Efficiently Anywhere:
Tools for Lock-free Concurrent Algorithms

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

John Bender

2019

© Copyright by

John Bender

2019

ABSTRACT OF THE DISSERTATION

Prove Once, Run Efficiently Anywhere:
Tools for Lock-free Concurrent Algorithms

by

John Bender
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2019
Professor Jens Palsberg, Chair

The multi-core revolution has pushed programmers and algorithm designers to build algorithms that leverage concurrency. This notoriously difficult task is futher complicated by the existence of weak architecture and language memory models. The presence of many such memory models has traditionally forced correctness proofs for lock-free concurrent algorithms to be performed on a per-model basis, resulting in a significant duplication of effort. We demonstrate that the correctness of lock-free concurrent algorithms can be proved once for implementations that can be compiled to run correctly and efficiently on all mainstream memory models.

The dissertation of John Bender is approved.

Lei He

Todd Millstein

Miryung Kim

Jens Palsberg, Committee Chair

University of California, Los Angeles

2019

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

The multi-core revolution has increased the need for shared-memory concurrent programming, algorithms, and languages. In response, programmers target multiple cores to speed up execution, algorithm designers develop concurrent libraries, and language designers make these tasks as easy as possible. Despite much progress, achieving general correctness results remains a challenge, because, in addition to reasoning about the nondeterminism of concurrent programs, a programmer must also reason in the context of a *memory model*.

Intuitively, a memory model is an interface that presents an idealized computer architecture, amenable to reasoning about correctness. A proof of correctness *assumes* a memory model, and then tool support *enforces* that memory model on a real architecture. This separation of concerns, proof from implementation, can simplify the proofs and make them more general, decreasing duplication of effort.

For example, the work of [35] assumes that the memory model is Sequential Consistency and proves the correctness of many algorithms. Then a tool made to support Sequential Consistency can enforce correctness on many architectures by, for example, inserting fences [4]. Similarly, Java algorithms rely on volatile variable annotations that the Java virtual machine enforces using fences [50]. Independently, many algorithms have been proven correct for the C/C++ memory models [81] that are enforced by compilers such as GCC and Clang.

However, this separation of concerns can also negatively impact performance. For example, many of the proofs in Herlihy and Shavit's book remain valid for memory models that are "weaker" than Sequential Consistency. The reason is that each of those proofs relies on a few assumptions, particular to the algorithm, that are implied by Sequential Consistency. The same is true of the relatively weak release-acquire fragment of C++ for certain algorithms [19]. Enforcing

these stronger assumptions is unnecessary and this translates into extra synchronization which hurts performance.

All this leads us to ask: *can we get performance, generality, and correctness?* Indeed, we will demonstrate that the correctness of lock-free concurrent algorithms can be proved once for implementations that can be compiled to run correctly and efficiently on all mainstream memory models. Specifically, we will describe the tools and techniques we have developed toward that end; including a compiler, the first memory model for Java's Access Modes [52, 51, 41], and a mechanized logic. In the spirit of Java's slogan "write once, run anywhere" and in recognition of our model of Java's Access Modes which forms the basis for our verification efforts, we say that algorithm designers can now "prove once, run efficiently anywhere."

The key to our approach is the use of *specified orders*. The idea is that the correctness of lock-free concurrent algorithms frequently turns on the ordered execution of a few critical instructions regardless of the target architecture or language.

To begin, we will show how one can arrive at the idea of specified orders naturally from an understanding of the relationship between sequential consistency and weaker memory models. First, we will describe sequential consistency informally. Then we will make the definition more precise so that we can accurately describe what is "weakened" in a weak memory model. Then we will see that specified orders arise from a desire to recover some of the ordering guarantees made by sequential consistency while benefiting from the performance of weaker memory models. Throughout we will use simple example programs to illustrate key ideas. Finally, we will conclude with an outline for how we will use specified orders to get fast, general, and correct algorithms in the rest of this work.

1.1 Sequential Consistency

Sequential consistency (SC) is the model that most programmers think of when they are reasoning about concurrent programs that access shared memory [48]. Conceptually, under SC, the processor chooses a thread from a set of threads defined by the running program. The processor then executes the next, unexecuted instruction from the chosen thread; here we are primarily

concerned with memory access instructions. The effect of each such memory access is recorded in main memory before choosing the next thread and memory access to execute. Finally, reads can look at main memory for a given location and produce the current value.

To see how reasoning proceeds under SC, consider the program in Figure 1.1. This program represents a standard message passing idiom in concurrent programming. The program performs two writes, `[x] := 1` and `[y] := 1`, to the locations x and y, concurrently with two reads, `[y]` and `[x]`, of the same locations into the registers `r1` and `r2`. The idea is, when the read of y in the second thread sees the value 1 it's a signal that some work has completed before the write to y in the first thread. In this case the "work" is the write of 1 to x. As an example, this technique is commonly used when constructing locks, where the write to y would be an unlock signaling that the lock of y is available to be acquired. Then, any process that acquires the lock can be assured that the work done in the critical section before the unlock has been written to memory.

```
[x] := 1  ||  r1 := [y]
[y] := 1  ||  r2 := [x]
```

Figure 1.1: Message Passing

We can phrase the desired behavior for this program as a question and use our understanding of sequential consistency to formulate an answer. The question is, if the read of y sees the value 1, then will the read of x see the completed write to x and the value 1? If we know that the read of y saw 1 we know that the write to y was executed. Intuitively, that means the write to x must have been executed since it comes before the write to y. From this we can conclude that the read of x sees the value 1 written by the write to x.

We can make this kind of reasoning more concrete by formalizing SC as three guarantees. These guarantees take the form of relationships between memory accesses in program executions.

1. There exists a total order over all executed memory accesses. Intuitively, the total order represents the order in which memory accesses are executed as the processor chooses a thread and an access.

2. The total order must be consistent with the order of instructions as defined in the program (program order). This represents the fact that the "next" instruction is determined by the sequence of the accesses in the chosen thread.

3. When a read is paired with a write to the same location (reads-from), the write must be the latest for that location in the total order. This represents the ability of a read to inspect the current state of global memory to find the latest value written to a given location.

Now we can ask questions about programs and use the formal definitions to provide precise answers. Consider the example execution in Figure 1.2 for the message passing program. Here, we represent an execution as a graph. Memory accesses are labeled nodes with a type of operation, a memory location, and a value. For example, the write to x of the value 1 labeled a appears as `a: W(x, 1)`. The relationships between accesses are labeled edges. Program order is represented as $\xrightarrow{\text{po}}$ and reads-from is represented as $\xrightarrow{\text{rf}}$. Any accesses not related by program order are assume to be in different threads.

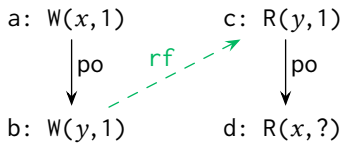


Figure 1.2: MP Graph

Now the question is, for executions where the message has been passed, b $\xrightarrow{\text{rf}}$ c, will d see the work performed by a? In the graph we can see that a $\xrightarrow{\text{po}}$ b $\xrightarrow{\text{rf}}$ c $\xrightarrow{\text{po}}$ d. Since the total order for execution guaranteed by SC must be consistent with these edges by guarantees 2 and 3, we know that a is executed before d in the total order. Thus, a is the latest write to x and d must see a and read the value 1 by guarantee 3.

1.2 Sequential Inconsistency

Much to the consternation of programmers [46] and researchers [62, 58], mainstream processors and languages do not have SC semantics. In the interest of improving performance they provide only a small subset of the relationships guaranteed by SC. For example, ARM processors may execute accesses to different locations out of program order.

To see the effect this has, we return to the message passing example in Figure 1.2. Under a weak memory model, like ARM, the order in which memory accesses are executed is *not* guaranteed to be consistent with program order. In particular we are no longer guaranteed that the edges a $\xrightarrow{\text{po}}$ b and c $\xrightarrow{\text{po}}$ d will affect the order in which the related accesses are executed. So,

even though the message may be received, b $\xrightarrow{\texttt{rf}}$ c, we can't conclude that d will see the value 1 as written by a. For example, the access a could be executed after b on ARM processors. Similarly, d could be executed before c.

1.3 Specified Orders

We can see in the message passing example that there are two ordering relationships guaranteed by SC that are required to ensure the correct behavior, namely the program order relationships a $\xrightarrow{\texttt{po}}$ b and c $\xrightarrow{\texttt{po}}$ d. Importantly, these are the *only* relationships that are required and we call these relationships specified orders. Other guarantees made by SC that are unrelated to the correctness of a given algorithm can be sacrificed in the name of performance.

Consider the extension to the message passing example in Figure 1.3. The specified orders now appear in blue along side the program order edges they are intended to recover. Further, there is now another write, e, to different variable z in the first thread. Recall that our only concern in message passing is that memory accesses before the write to y in the program are completed before memory accesses after the read of y in the program, specifically the write to x and the read of x respectively.

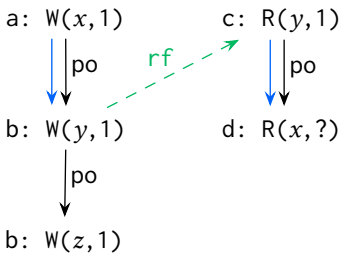


Figure 1.3: MP Extra Access

The desired behavior does *not* depend on the write to z executing in any particular order and it can freely be reordered before the other accesses in the first thread. That is, the guarantee that e executes after a and b provided by SC is unnecessary. This allows us to benefit from any performance increase we might see by reordering those accesses and our algorithm will continue behave correctly [1].

[1]When the message passed is an unlock for the variable this freedom to reorder accesses like the write to z is known as roach motel reordering [77]. The idea is that any access outside a critical region in the program can be reordered to execute inside the critical section without issue.

1.4 In Theory and in Practice

Specified orders are an intuitive response to the lack of guarantees provided by modern hardware and compilers. However, we will demonstrate that they can be more than an idealized form of synchronization. Indeed, we will show that they enable us to construct single algorithm that can be compiled to fast, practical code for many architectures.

In Section 2 we demonstrate the practicality of specified orders. We will show that specified orders can be compiled to efficiently synchronized code for the x86 and ARMv7 architectures. In Section 3 we build a formal model for Java's Access Modes [52, 51] and demonstrate that the core of the model admits more weak memory model behaviors than any other mainstream memory model. We will also show why this makes it an ideal target for general reasoning. In Section 4 we build on our formal model of Java's Access Modes and detail a logic we have constructed that we use in mechanized correctness proofs of lock-free algorithms with specified orders. We demonstrate the capabilities of the logic with proofs for a ring buffer as found in the Linux kernel [38] and Dekker's mutual exclusion algorithm. Finally, in Section 5 we place this work in the broader context of research for verification under weak memory models.

CHAPTER 2

Fast Code from Specified Orders

While specified orders are a natural extension to formal relational models of weak memory, they are more than just a theoretical concern. Here, we will demonstrate that specified orders are practical and result in *fast* code. Specifically, we will show that memory fences can be inserted intelligently to enforce specified orders for many processor architectures and the resulting code is competitive with hand written equivalents.

In Section 2.1 we will give an overview of the challenges inherent in inserting fences to enforce specified orders. In Section 2.2 we will detail our algorithm for inserting fences. In Section 2.3 we will provide implementation details for our fence insertion tool, Parry, which implements the algorithm. In Section 2.4 we demonstrate that our approach results in code that is as fast as code written by experts.

2.1 Fence Insertion

To enforce the correct ordering of memory accesses architectures provide memory fence instructions. These instructions guarantee that some subset of the supported memory operations will take place before and after the fence during execution. For example, on ARM processors, the `dmb st` fence guarantees that all stores before the fence in program order will complete before the fence is finished executing, thereby ordering those stores with instructions after the fence. In the case of the message passing example in Figure 1.2 we can insert a `dmb st` fence between `a` and `b` to ensure that the store to x completes before the store to y.

The problem of enforcing specified orders becomes more complicated when considering portability between architectures. For the message passing example, the specified orders require no

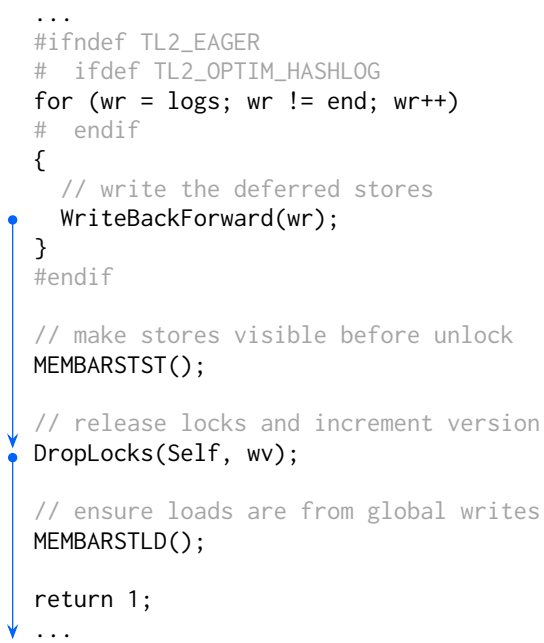


Figure 2.1: STAMP TL2 `TxCommit` Procedure

fences on x86. On ARM processors the specified order in the first thread can be enforced by either a `dsb`, `dmb`, or a `dmb st`. Similarly the specified order in the second thread can be enforced by either a `dsb`, `dmb`, or a `dmb ld`. These choices have an impact on the performance of the algorithm during execution. We call these choices *fence selection*. Additionally we consider the placement or insertion of fences. For the message passing example the placement is straightforward but this is not always the case.

2.1.1 TL2 Commit

We will illustrate the complexity of *fence insertion* by examining the commit procedure of the TL2 transactional memory algorithm which we will reference throughout this section.

Intuitively, a transactional memory [34, 79] is a concurrent object that encapsulates and manages accesses to an array of memory locations. The TM interface has four highly concurrent methods, namely init, read, write, and commit, that a typical user program calls a large number of times.

When TL2 is managing a transaction, stores made inside the transaction do not go to main memory. Instead TL2 records the stores in a "write-set". When the transaction ends, the algorithm attempts to acquire locks for each address, commit the write-set to main memory, and release the acquired locks.

For TL2's commit to function properly the "real" stores made to each memory address from the write-set must be seen to take place before the release of the corresponding locks. Otherwise, an external observer may see an address in the write-set as unlocked before the actual store from the write-set makes it to main memory.

Similarly, the release of the lock for each address must be seen to take place before any load after the commit is finished. This ensures that loads performed in the same thread as the transaction will see the same values in memory from the write-set and locks as any external observer.

Figure 2.1 shows the source code for these orders from the TL2 commit procedure, `TxCommit`, as it appears in implementation included with the STAMP benchmark suite [24]. The `WriteBackForward` procedure contains the store instruction that moves values from the write-set to main memory and the `DropLocks` procedure contains the store instruction that releases the locks.

We record the specified orders from `WriteBackForward(wr)` to `DropLocks(Self, wv)` and from `DropLocks(Self, wv)` to all later loads (`...`) as blue arrows. To enforce these orders, the TL2 designers have placed memory fence macros between the relevant operations. An implementer who is porting TL2 to different architectures can define each macro to be an architecture appropriate memory fence to enforce the correct behavior. The drawback of such a fence-centric approach is that for a programmer who wishes to understand the TM algorithm and perhaps to port it to a different architecture, a fence placement says little about why the designers chose it and placed it at a particular program point. Fences are best viewed as an implementation mechanism for a higher level of abstraction.

Fence Insertion. Inserting fences requires knowledge of the control flow paths between the ordered instructions, as well as the other instructions on those paths.

Without knowing the control flow information it is possible to miss paths and allow executions in which the instructions may be seen to pass each other. On the other hand, a naive

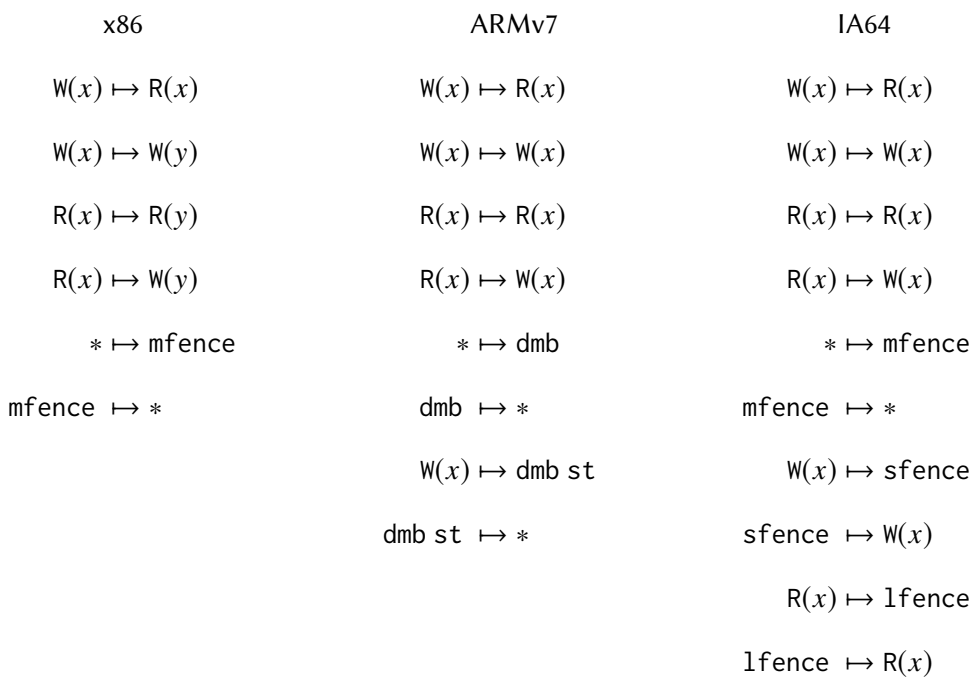


Figure 2.2: Architecture Definitions

approach to fence placement that avoids missing paths by inserting a fence directly after the first instruction in the order can be expensive. For example, if the first fence macro is placed directly after the call to `WriteBackForward` it can result in an expensive loop over the fence when the `TL2_OPTIM_HASHLOG` flag is set at compile-time.

Without knowing the other instructions in the control flow paths, one might place a new fence where another already exists, or where properties of the memory models makes fences unnecessary in the presence of other instructions.

For example, consider the first order in `TxCommit`. The x86 memory model already enforces many orders. Looking at the orders enforced by the x86 architecture definition in Figure 2.2, we can see $\mathsf{W}(x) \mapsto \mathsf{W}(y)$ suggests that two stores to any address will not move past each other. We use this to prove that the instructions in the first order of `TxCommit`, $\mathsf{W}(\mathsf{addr}) \rightarrow \mathsf{W}(\mathsf{lock})$, will not move past each other:

$$p, \mathsf{x86} \vdash \mathsf{W}(\mathsf{addr}) \rightarrow \mathsf{W}(\mathsf{lock})$$

$$\dfrac{\dfrac{}{p, \mathsf{x86} \vdash \mathtt{W}(\mathsf{lock}) \longrightarrow \mathtt{W}(\mathsf{tmp})} \qquad \dfrac{\dfrac{}{p, \mathsf{x86} \vdash \mathtt{W}(\mathsf{tmp}) \longrightarrow \mathtt{R}(\mathsf{tmp})} \quad \dfrac{}{p, \mathsf{x86} \vdash \mathtt{R}(\mathsf{tmp}) \longrightarrow \mathtt{R}(x)}}{p, \mathsf{x86} \vdash \mathtt{W}(\mathsf{tmp}) \longrightarrow \mathtt{R}(x)}}{p, \mathsf{x86} \vdash \mathtt{W}(\mathsf{lock}) \longrightarrow \mathtt{R}(x)}$$

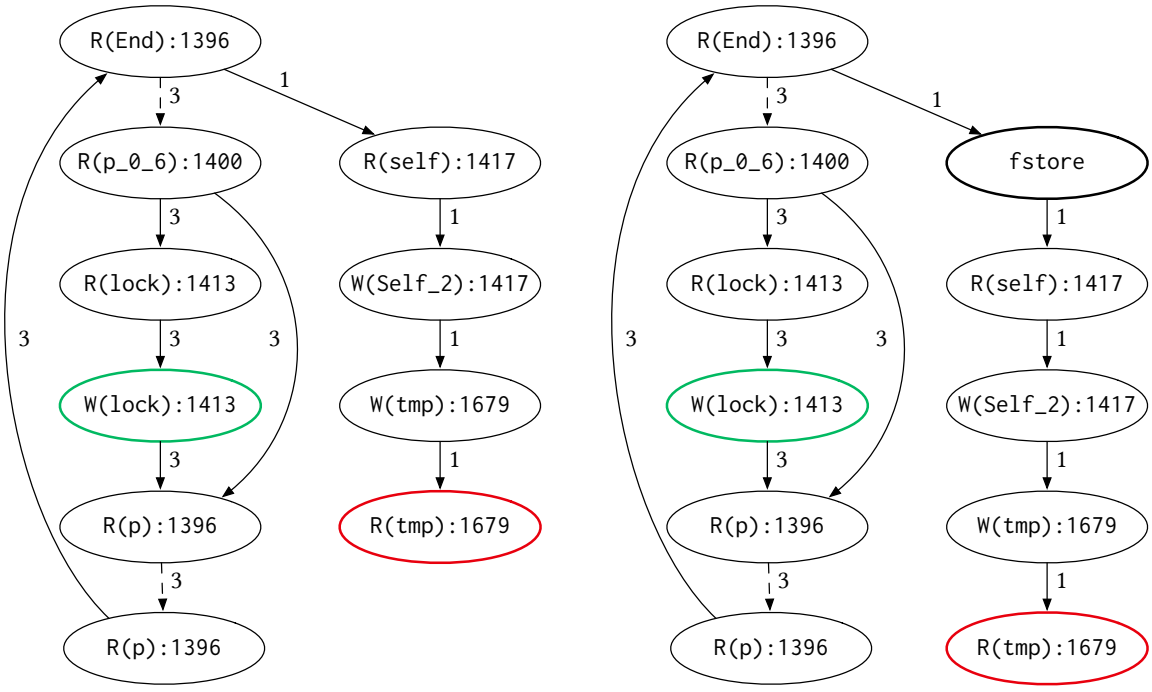
Figure 2.3: Derivation of $\mathtt{W}(\mathsf{lock}) \longrightarrow \mathtt{R}(x)$ in TxCommit

That is, if x86 prevents stores from moving past each other and the order we want to enforce involves two stores, then we can conclude that the order is enforced. This means we can safely define the first macro in the example as a no-op on x86. Note that our architecture rules for x86 do not include non-temporal hinted store instructions like `movnti` and `movntdq`.

In contrast, this order is not enforced by ARMv7. Since stores can be seen to move past other stores when the addresses are different on ARMv7 we do not include this rule in the architecture definition. Instead the rule relating stores requires that the addresses be the same.

The second order in `TxCommit`, $\mathtt{W}(\mathsf{lock}) \longrightarrow \mathtt{R}(x)$, represents a more complex example of how intervening instructions can affect order enforcement. When `TxCommit` is compiled with Clang, the compiler generates a store and a load to the same temporary address after the lock release in `DropLocks` but before the end of the procedure as illustrated in both graphs in Figure 2.4. We can use these instructions and the properties of x86 to prove the *transitive order* in Figure 2.3 where tmp represents the temporary memory location. We conclude that the store to lock can never be seen to move past the final load in `TxCommit` and also any subsequent load. We define the notation, rules and memory model properties more completely in the next section.

Selecting Fences. Selecting the correct fence requires knowledge of how the compiler will treat the source code and knowledge of the fences available for each architecture. For `TxCommit` on ARMv7 the second macro can be defined correctly using many different fence configurations according to the ARM documentation [6], e.g. `dsb`, `dmb`, or a qualified `dmb st`. Both the litmus test documentation [32] and the assembler reference [6] are complicated texts in accordance with the complexity of the ARMv7 memory model. Determining the best fence is a nontrivial task.



(a) Before (b) After

Figure 2.4: Fence Insertion for a Modified Control Flow Graph

2.1.2 Our Approach

We decompose fence insertion into two sub-problems. First we eliminate any orders that are provably enforced by existing instructions and the properties of the target architecture. For the remaining orders we modify the program with new instructions that enforce the remaining orders.

We address both sub-problems by considering control flow graphs with a restricted set of instructions. Each node is labeled with the instruction, the address that it operates on, and the line number that it was generated from in the source code. Every path between two instructions in the control flow graph represents a possible execution involving those instructions. We construct these graphs using the intermediate representation and control flow graph for a given procedures as generated by the LLVM compiler [49].

In Figure 2.4a we have a control flow sub-graph for the second order in `TxCommit`, constructed from LLVM's output when compiling the procedure. It contains all of the paths and a subset of

the instructions (elided with dashed arrows) that appear between the store, W(lock), to release the locks in DropLocks and the end of the TxCommit procedure. Note that the store to release the locks appears in a cycle that comes from the body of the DropLocks procedure though it does not appear in the code in Figure 2.1. Also the name of the variable containing the lock address has been renamed to lock for clarity.

Order Elimination. We eliminate an order by proving that the order is enforced for every path between the ordered instructions. Proofs for orders correspond with paths in the graph through a second set of *architecture edges*. If two instructions exist in a control flow path and the architecture guarantees that they will never move past each other, we add an architecture edge.

Returning to our example, recall that on x86 we can eliminate the second order in TxCommit because the compiler generates a load and store to a temporary variable. In Figure 2.4a, we add architecture edges (not pictured) between two nodes pairs. We add the first edge between the nodes labeled with W(lock):1413 and W(tmp):1679 because stores cannot move past stores. We add the second edge between the nodes labeled with W(tmp):1679 and R(tmp):1679 because stores can not move past loads from the same address. These edges correspond with a transitive order derivation for every path between the store to lock and the load to tmp. As a result we can eliminate the order.

In contrast, on ARMv7, we cannot add the first architecture edge between the store to lock and the store to tmp because stores are permitted to move past other stores for different addresses. In that case, since we can't eliminate the order with the original graph, we must alter the graph so that we can prove the order is enforced.

Fence Insertion. We model the problem of finding these graph alterations as minimum multi-commodity cut [18] (hereafter multi-cut). Intuitively, multi-cut finds a minimum set of edges such that, when they are removed, no paths exist between the sources and sinks for all commodities. If we define the sources and sinks for commodities using the paired instructions in orders and then "split" the edges in the resulting cut with a fence, we will have transitive orders for all paths and all orders.

The altered control flow graph in Figure 2.4b shows the results of fence insertion on ARMv7

for `TxCommit`. The algorithm selects a single edge from the original graph to split with a fence, represented here as fstore. This alteration ensures that all paths from the store to the load are provably enforced.

Note also that the cut should happen outside the loop-cycle in the control flow graph. This prevents an unnecessary performance penalty when placing the fence. This is handled directly by the minimum cut algorithm. Since the cut is determined based on the sum of the capacities of the edges in the cut, we can use larger capacities to discourage the selection of edges that occur in loops. This ensures a fence will only be placed in a loop when all paths for an order are in a loop. Modeling fence insertion as multi-cut accounts for the full generality of control flow graphs including odd control flow configurations and order overlap.

Fence Selection. Finally, we select appropriate fences for each placement by defining a partial order over fences based on their capabilities. For example on ARMv7 a `dmb` fence can enforce strictly more orders than `dmb st` since the latter only waits for stores. In the case of Figure 2.4 either fence will work but we prefer the "weaker" `dmb st` represented here as the abstract fence type fstore. We do this under the assumption that it is less costly during execution which is supported by the ARMv7 documentation [32].

2.1.3 Orders, Not Fences

The memory fence is a blunt instrument that relates possibly hundreds of instructions across many execution paths and blurs its original purpose. Instead we supply a scalpel in the form of specified orders, which are more specific about the desired behavior of the program. Specified orders at the source level enable authors to reason more easily about their algorithm, while a compiler can do the work to insert fences that enforce the orders.

2.2 The Fence Insertion Algorithm

Our algorithm for fence insertion takes three inputs: a control-flow graph G, architecture rules A, and specified orders O. The output is a transformed graph $\mathsf{Insert}(G, A, O)$ with fences that

enforce the specified orders. In Section 2.2.1, we define a basic version of Insert and prove it correct. The basic version relies on the simplifying assumption that the only fence available is fany. We will assume that fany enforces that all instructions prior to the fence happen before all instructions after the fence. In Section 2.2.2, we describe briefly how to generalize Insert to work with multiple fences.

2.2.1 The Core Algorithm

We proceed as follows. First we define the basic concepts of graphs, architecture rules, and specified orders, along with some helper notation. Then we define a correctness criterion of the form $G, A \models O$, which says that the combination of G and A enforces all the specified orders O. This brings us to definition of our algorithm Insert whose goal is to produce an output graph G that satisfies $G, A \models O$. Finally, we give an example and then prove the correctness of Insert.

Graphs. A control-flow graph $G = (V, E, \ell)$ consists of a set V of nodes, a set $E \subseteq (V \times V)$ of directed edges, and a labeling function ℓ. Intuitively, a node is a program point, the label of a node is the instruction at that program point, and an edge is potential control flow between two program points. We use i, j to range over V. The function ℓ maps each element of V to a *label*, which is an element of

$$\{\ \mathtt{W}(a),\ \mathtt{R}(a),\ \mathsf{fany}\ \}$$

where a is an address, W represents a *store*, R represents a *load*, and fany is a fence. We use l to range over labels. Notice that our control-flow graphs focus entirely on loads, stores, and fences. This is is in contrast to the conventional notion of a control-flow graph that represents every instruction in a program. One can abstract such a conventional graph into one of our graphs by, intuitively, omitting the nodes of no interest to our approach.

For a graph G with i_1, i_2 among its nodes, we use $\mathsf{paths}(G, i_1, i_2)$ to denote the set of *paths* in G from i_1 to i_2. A path from i_1 to i_2 is itself a graph in which 1) each node has one outgoing edge, except i_2 which has no outgoing edges, and 2) all nodes on the path are reachable from i_1 by following zero, one, or more edges. Our notion of path is usually known as a *simple path* because it allows no loops. Still, we will use the terminology "path" for simplicity. We will use p to range

over paths.

Architecture rules. A set of architecture rules specifies a memory model. Intuitively, the fewer the rules, the weaker the memory model. The idea is that even if a control-flow graph has an edge from i_1 to i_2, the execution of i_1 and i_2 may happen in either order or overlap, unless specific architecture rules enforce an order of execution. A set A of architecture rules consists of rules of the form $L \mapsto R$, where L, R are rule components that range over

$$\{ *,\ \mathtt{W}(x),\ \mathtt{R}(x),\ \mathsf{fany} \}$$

and where x is a variable that ranges over addresses. Intuitively, $*$ is a wildcard. A rule $L \mapsto R$ expresses that if we have a graph (V, E, ℓ) with two nodes i_1 and i_2 such that i_1 can reach i_2, and such that we can *instantiate* $L \mapsto R$ to $(\ell(i_1), \ell(i_2))$, then we can conclude that i_1 must happen before i_2. We will define the notion of instantiation below.

For example, the rules $(* \mapsto \mathsf{fany})$, $(\mathsf{fany} \mapsto *)$ express, intuitively, that fany is a fence. Specifically, the first rule says that all instructions that can reach the fence will happen before the fence, while the second rule says all instructions that can be reached from the fence will happen after the fence. The combined effect of those two rules is that all instructions prior to the fence happen before all instructions after the fence. In this section we define Insert in a way that relies on that A contains those two rules.

As another example, the rule $\mathtt{W}(x) \mapsto \mathtt{W}(y)$ expresses, intuitively, that all store instructions must happen in the order in which they are reached in the control-flow graph.

As a third example, the rule $\mathtt{W}(x) \mapsto \mathtt{R}(x)$ expresses, intuitively, that if a store instruction to a particular address can reach a load instruction from that same address in the control-flow graph, then the store instruction must happen before the load instruction.

We will now define a notion of *instantiating* an architecture rule to a pair of labels. Specifically, if $(L \mapsto R)$ is an architecture rule and (l_1, l_2) is a pair of labels, then we write $(L \mapsto R) \triangleright (l_1, l_2)$ to denote that $(L \mapsto R)$ instantiates to (l_1, l_2).

The definition of instatiation will ensure that for rules such as $(\mathtt{W}(x) \mapsto \mathtt{R}(x))$, the two occurrences of x must be replaced with the *same* address. Our technical device to make that happen is that of a *substitution*. We use σ to range over substitutions that map variables of the form

x to addresses. For our use, each substitution has either a domain of either zero, one, or two elements, depending on whether a rule mentions zero, one or two variables. The definition of $(L \mapsto R) \triangleright (l_1, l_2)$ uses the relation $\blacktriangleright$ to distribute the use of a substitution to each of L and R. Now we are ready to present the detailed definition of instantiation.

We say that a rule $(L \mapsto R)$ instantiates to a pair of labels (l_1, l_2) if we can derive $(L \mapsto R) \triangleright (l_1, l_2)$ with the following rules:

$$\frac{(L, \sigma) \blacktriangleright l_1 \qquad (R, \sigma) \blacktriangleright l_2}{(L \mapsto R) \triangleright (l_1, l_2)}$$

$$\begin{aligned}
(*, \sigma) &\blacktriangleright l \\
(\mathtt{W}(x), \sigma) &\blacktriangleright \mathtt{W}(\sigma(x)) \\
(\mathtt{R}(x), \sigma) &\blacktriangleright \mathtt{R}(\sigma(x)) \\
(\mathsf{fany}, \sigma) &\blacktriangleright \mathsf{fany}
\end{aligned}$$

The first rule says that we can instantiate $(L \mapsto R)$ to (l_1, l_2) if we can find a substitution σ such that L guided by σ instantiates to l_1 (written $((L, \sigma) \blacktriangleright l_1)$), and R guided by σ instantiates to l_2 (written $((R, \sigma) \blacktriangleright l_2)$). The other four rules define the cases where a rule component, guided by a substitution, instantiates to a label. Specifically, $*$ instantiates to any label, $\mathtt{W}(x)$ instantiates to $\mathtt{W}(\sigma(x))$, $\mathtt{R}(x)$ instantiates to $\mathtt{R}(\sigma(x))$, and fany instantiates to fany.

Specified Orders. For a graph $G = (V, E, \ell)$, the specified orders is a set $O \subseteq (V \times V)$.

Correctness criterion. We will now define a correctness criterion $G, A \models O$. Intuitively, $G, A \models O$ says that the combination of G and A enforces all the specified orders O. The goal of our approach is to produce an output graph G that satifies $G, A \models O$.

We define the correctness criterion in two steps. First we define a judgment $p, A \vdash i_1 \rightarrow i_2$. For a path $p = (V, E, \ell)$, architecture rules A, and nodes i_1, i_2 on p, we define that $p, A \vdash i_1 \rightarrow i_2$

holds if it can be derived by these rules:

$$p, A \vdash i_1 \rightarrow i_2 \quad \begin{array}{l}(\text{where } i_1 \text{ can reach } i_2 \text{ in } p \wedge \\ \qquad (L \mapsto R) \in A \wedge \\ \qquad (L \mapsto R) \rhd (\ell(i_1), \ell(i_2)) \ )\end{array}$$

$$\frac{p, A \vdash i_1 \rightarrow j \qquad p, A \vdash j \rightarrow i_2}{p, A \vdash i_1 \rightarrow i_2}$$

The first rule instantiates an architecture rule in A, and the second rule is transitivity.

Now we are ready to define the overall correctness criterion. For a graph $G = (V, E, \ell)$, architecture rules A, and specified orders $O \subseteq (V \times V)$, define:

$$\begin{array}{l} G, A \models O \iff \\ \forall (i_1, i_2) \in O : \ \forall p \in \mathsf{paths}(G, i_1, i_2) : \ p, A \vdash i_1 \rightarrow i_2 \end{array}$$

Notice that the definition considers *all* paths between i_1 and i_2. This ensures that the specified order will be enforced, irrespective of the control flow.

Algorithm overview. Our algorithm Insert composes three functions Elim, Cut, and Refine. Intuitively, Insert proceeds in three steps:

1. Elim determines a subset of the specified orders that are enforced by the architecture.
2. Cut determines *where* to insert fences that will enforce the rest of the specified orders.
3. Refine inserts the fences.

We now describe Elim, Cut, Refine, and Insert. After those descriptions, we will give an example.

The Elim function. Our approach uses a function Elim that determines a subset of the specified orders for which we need *no* fences. We rely on the fact that Elim satisfies the following property:

$$\mathsf{Elim}(G, A, O) \subseteq O \text{ and } G, A \models \mathsf{Elim}(G, A, O). \tag{2.1}$$

Programmers can implement $\mathsf{Elim}(G, A, O)$ in many ways, including the trivial approach that always returns the empty set. Our implementation, as a default, uses a straightforward exponential-time algorithm that for each $(i_1, i_2) \in O$ enumerates all $p \in \mathsf{paths}(G, i_1, i_2)$, and for each such p uses brute-force to determine whether $p, A \vdash i_1 \rightarrow i_2$. The result is that $\mathsf{Elim}(G, A, O)$ returns a maximal subset of O. The maximal size helps us insert few fences.

In addition we have implemented a linear time approximation algorithm which works by finding enough nodes i with the property, $\{(i_1, i), (i, i_2)\}, A \vdash i_1 \rightarrow i_2$ such that, when every i is removed $\mathsf{paths}(G, i_1, i_2) = \emptyset$.

In either case, we need no modifications to G to enforce the orders in $\mathsf{Elim}(G, A, O)$ so now let us focus on where to insert fences to enforce the orders in $O \setminus \mathsf{Elim}(G, A, O)$.

The Cut function. Our approach uses a function Cut that determines *where* to insert fences. We rely on that Cut satisfies the following property:

$$\mathsf{Cut}(G, O) \text{ is a } \textit{multi-cut} \text{ for } G, O. \tag{2.2}$$

The multi-cut specifies where to insert fences. Let us recall the standard notion of a multi-cut [18]: given a graph $G = (V, E, \ell)$ and a set $O \subseteq (V \times V)$, a multi-cut for G, O is a set K, where $K \subseteq E$, such that $\forall i_1, i_2 \in O : \mathsf{paths}((V, E \setminus K, \ell), i_1, i_2) = \emptyset$. Programmers can implement $\mathsf{Cut}(G, O)$ in many ways, such as the trivial approach that always returns E, an approximation algorithm [14], and an integer linear program [14]. We experimented with those and chose an ILP with a polynomial number of constraints in the size of the graph [14]. We use SAGE [73] and the default solver GLPK [30] to solve the ILP, which returns a multi-cut of minimal size, which in turn helps us insert few fences. Given G, O and an integer n, the problem to decide whether there exists a multi-cut for G, O with at most n elements is NP-complete for $|O| > 2$ [18]. Now let us consider *how* to use a multi-cut to insert fences.

The Refine function. Our approach uses a function Refine that inserts fences. We will give the definition of Refine in detail and later we will prove that the definition satisfies four lemmas. For a graph $G = (V, E, \ell)$ and a cut-set K, where $K \subseteq E$, the function $\mathsf{Refine}(G, K)$ creates a set W_K of additional nodes (fences!), and replaces each $(j_1, j_2) \in K$ with two new edges

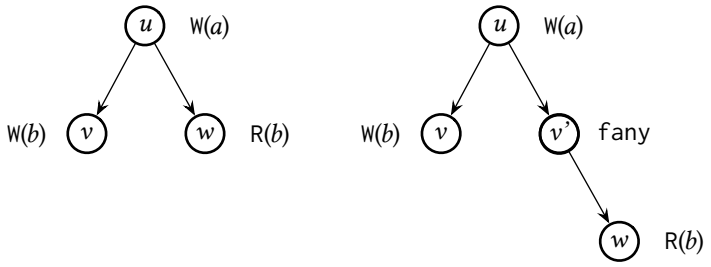


Figure 2.5: An example graph and its refinement.

that, intuitively, insert a fence between j_1 and j_2. The new nodes form a set W_K:

$$W_K \quad ::= \quad \{v_{j_1,j_2} \mid (j_1, j_2) \in K\}$$

where each v_{j_1,j_2} is a fresh node. The output graph is:

$$\begin{aligned} \mathsf{Refine}(G, K) &= (V \cup W_K,\ (E \setminus K) \cup E_K,\ \ell_K) \\ E_K &= \{(j_1, v_{j_1,j_2}), (v_{j_1,j_2}, j_2) \mid v_{j_1,j_2} \in W_K\} \\ \ell_K &= \ell \cup \{(v_{j_1,j_2}, \mathsf{fany}) \mid (j_1, j_2) \in K\} \end{aligned}$$

Notice that $\mathsf{Refine}(G, \emptyset) = G$.

The Insert function. We can now define Insert:

$$\mathsf{Insert}(G, A, O) \quad = \quad \mathsf{Refine}(G,\ \mathsf{Cut}(G, O \setminus \mathsf{Elim}(G, A, O)))$$

The definition calls three functions as outlined above: first Elim, then Cut, and finally Refine. Both Elim and Cut run in worst-case exponential time, while set difference and Refine run in polynomial time, so we conclude that Insert runs in worst-case exponential time.

Example. Consider the graph $G = (V, E, \ell)$, which is illustrated in Figure 2.5 (left graph). We have

$$\begin{aligned} V &= \{u, v, w\} & \ell(u) &= \mathtt{W}(a) \\ E &= \{(u, v), (u, w)\} & \ell(v) &= \mathtt{W}(b) \\ & & \ell(w) &= \mathtt{R}(b) \end{aligned}$$

where a, b are distinct addresses. Consider also the set of architecture rules:

$$\begin{aligned} A \quad = \quad & \{(* \mapsto \mathsf{fany}),\ (\mathsf{fany} \mapsto *), \\ & (\mathtt{W}(x) \mapsto \mathtt{W}(y)),\ (\mathtt{W}(x) \mapsto \mathtt{R}(x))\} \end{aligned}$$

Consider finally the specified orders

$$O \quad = \quad \{(u, v), (u, w)\}$$

A run of $\mathsf{Insert}(G, A, O)$ will proceed as follows.

The first step is to call $\mathsf{Elim}(G, A, O)$. This call to Elim will find that $G, A \models \{(u, v)\}$ because we have a single-edge path from u to v, and a rule $(\mathtt{W}(x) \mapsto \mathtt{W}(y)) \in A$ that instantiates to $(\ell(u), \ell(v))$, which is equal to $(\mathtt{W}(a), \mathtt{W}(b))$. Thus, if p is the single-edge path from u to v, we can derive $p, A \vdash u \rightarrow v$. The call to Elim will also find that $G, A \not\models \{(u, w)\}$ because we have no rule in A that for the single-edge path p' from u to w enables us to derive $p', A \vdash u \rightarrow w$. Note here that the rule $(\mathtt{W}(x) \mapsto \mathtt{R}(x)) \in A$ requires the two instructions (store and load) work with the same address, while the two labels $\mathtt{W}(a)$ and $\mathtt{W}(b)$ operate on distinct addresses. In summary, we have $\mathsf{Elim}(G, A, O) = \{(u, v)\}$. We can calculate $O \setminus \{(u, v)\} = \{(u, w)\}$.

The second step is to call $\mathsf{Cut}(G, O \setminus \mathsf{Elim}(G, A, O) = \mathsf{Cut}(G, O \setminus \{(u, v)\} = \mathsf{Cut}(G, \{(u, w)\})$. In this case we have $\mathsf{Cut}(G, \{(u, w)\}) = \{(u, w)\}$. The reason is that the one path from u to w has a single edge so we must have that edge in the multi-cut.

The third step is to call

$$\begin{aligned} \mathsf{Insert}(G, A, O) \quad &= \quad \mathsf{Refine}(G, \ \mathsf{Cut}(G, O \setminus \ \mathsf{Elim}(G, A, O))) \\ &= \quad \mathsf{Refine}(G, \{(u, w)\}) \end{aligned}$$

The result is illustrated in Figure 2.5 (right graph). Compared to G, the graph $\mathsf{Insert}(G, A, O)$ has an additional node v', no edge $\{(u, w)\}$, but instead two edges (u, v') and (v', w). We have $\ell(v') = \mathsf{fany}$, that is, the new node is a fence. We can instantiate the rule $(* \mapsto \mathsf{fany}) \in A$ to $(\mathtt{W}(a), \mathsf{fany})$, and we can instantiate the rule $(\mathsf{fany} \mapsto *) \in A$ to $(\mathsf{fany}, \mathtt{R}(b))$. So for the path p'' with the two edges (u, v') and (v', w), we have that we can derive $p'', A \models (u, w)$.

In summary, the example has two specified orders, and one of them is enforced by the architecture without insertion of any fences, while for the other, we inserted a single fence.

Theorem 1. *If* $\{(* \mapsto \mathit{fany}), (\mathit{fany} \mapsto *)\} \subseteq A$, *then* $\mathit{Insert}(G, A, O), A \models O$.

We prove Theorem 1 in Appendix B.

2.2.2 Multiple Kinds of Fences

Let us now relax the assumption that the only fence available is fany. For example, Figure 2.2 gives rules for the two fences on ARMv7, namely the weaker fence `dmb st` and the stronger fence `dmb`. When multiple fences are available, the Refine function can choose as weak a fence as possible. The idea is that a weaker fence executes faster than a stronger fence, which is true for the architectures we have considered. Intuitively, we let Refine choose the weakest fence that enforces the relevant declared executions orders.

We will explain how to make the choice in two steps. First let us consider a simple case, which happens to be the one we encountered exclusively in our experiments. For an edge (j_1, j_2) in a cut set K, suppose we have a single element $(i_1, i_2) \in O \setminus \mathsf{Elim}(G, A, O)$ for which (j_1, j_2) is on a path from i_1 to i_2. We must chose a fence that is strong enough to enforce the order (i_1, i_2). Specifically, we need a fence f such that A contains the rules $(L \mapsto f)$ and $(f \mapsto R)$ such that

$$(L \mapsto f) \rhd (i_1, f) \ \wedge \ (f \mapsto R) \rhd (f, i_2)$$

We will choose as weak a fence as possible. For the archictectures we have considered, we can always find a fence that is the weakest among all those that satisfy the above requirement.

Now let us consider the general case. For an edge (j_1, j_2) in a cut set K, suppose we have multiple elements $(i_1, i_2) \in O \setminus \mathsf{Elim}(G, A, O)$ for which (j_1, j_2) is on a path from i_1 to i_2. Here we want the least expensive fence that will enforce *all* of the involved orders. For each such (i_1, i_2) we chose a fence $f_{(i_1,i_2)}$ as described above. Now we need a fence that is at least as strong as those fences $f_{(i_1,i_2)}$. Again, for the archictectures we have considered, we can always find a weakest fence that is at least as strong as each $f_{(i_1,i_2)}$.

Optimality. The optimality of our approach to fence selection depends on the assumption that any two fences will always execute more slowly than any single fence. Consider the case of an architecture like IA64 in Figure 2.2 and some procedure where we have a load-load order and a store-store order overlapping on a single simple path. Individually the orders can be enforced with an `lfence` *and* an `sfence` but the optimal multi-cut will include an edge shared by both orders. Then to satisfy both orders with a single fence we must select an `mfence`.

2.3 Implementation

We have implemented our approach in a tool called Parry [11] that takes as input a concurrent algorithm written in C/C++, declared execution orders, and a memory model, and as output produces C/C++ with fences. Parry uses Python to orchestrate the three major tasks in fence insertion: control-flow graph generation, order elimination, and fence insertion. We will now explain some details of Parry, particularly a few points that go beyond the fence insertion algorithm that we described in Section 2.2.

Graph Generation Parry is based on LLVM. First Parry compiles the input source code to LLVM's static-single-assignment (SSA) intermediate representation (IR) along with debugging information. Then Parry generates a control-flow graph of the target procedure using LLVM's `opt` tool. Next, Parry simplifies the control-flow graph by replacing each standard block with a path of instructions. We manipulate the resulting graphs with the graph-tool library [23].

We construct a graph in which the only nodes are for load, store, call, and `cmpxchg` instructions. Note that compared to the algorithm in Section 2.2, we add call instructions to safely account for methods which do not get inlined, and we add the `cmpxchg` instructions because they are frequently used by authors to enforce orders. Indeed, the `cmpxchg` instruction provides compare-and-swap semantics at the LLVM IR level and can act as a full memory fence (like `mfence` on x86 or `dmb` on ARMv7) [72].

Compiler Assumptions Parry uses Clang to translate C/C++ into LLVM's intermediate representation and we assume that this translation preserves some key aspects of the code that are of interest to Parry.

We assume that the semantics of a line of C/C++ used to specify an order will be preserved in the intermediate representation generated by Clang. For example, if the programmer expects a store to a certain memory address at a line in the code then we assume that Clang will generate a store to that address for that line. We safely account for the possibility of more than one instruction per line matching the event types of an order by including all matching instructions

during the analysis.

We also assume that our fence placements will remain valid after a compiler optimizes the C/C++ code that Parry outputs. That is, we require that the ordered instructions will not be moved past the fences by the compiler. To that end we ensure that all inline assembly instructions inserted by Parry are marked as volatile operations.

Architecture Rules As detailed in Section 2.2 we have created a set of rules for each architecture that describe the which instructions won't "move" during execution. These rules are necessary for order elimination.

We have included three architectures in Figure 2.2, x86, ARMv7 and IA64. The last is for clarity and comparison since our evaluation does not include benchmarks for IA64.

We compiled these rules based on our interpretation of the processor documentation available for each architecture. They are intended to be an over-approximation of the actual architecture behavior. During order elimination they are used to establish preexisting orders without consideration for other types of instructions aside from stores, loads, and fences.

Not included in Figure 2.2 are instructions that exist in the LLVM IR, like `cmpxchg`, which result in hardware instructions with fence-like semantics. We do account for LLVM's `cmpxchg` as detailed in Section 2.4.7.

Edge Elimination Parry has an initial step that takes place before the main algorithm in Section 2.2: edge elimination. The idea is to eliminate all edges that are irrelevant to the fence insertion problem. We keep an edge only if for at least one declared execution order, the edge is on a path from the source to the sink of the order *and* the instructions on that path don't enforce the order. After edge elimination, we can implement order elimination for an order (i_1, i_2) simply by checking whether the set of paths from i_1 to i_2 is empty.

Address Equality In some cases, an architecture rule uses a variable twice, such as the rule $(\mathtt{W}(x) \rightarrow \mathtt{R}(x))$. Our tool only instantiates the rule in case x can be replaced with a variable in LLVM's internal static-single-assignment form. The SSA form guarantees that the value of that

variable is the same at both program points. For example, in Figure 2.4, `tmp` is a variable, so we can instantiate $(\mathtt{W}(x) \rightarrow \mathtt{R}(x))$ to $(\mathtt{W}(\mathtt{tmp}), \mathtt{R}(\mathtt{tmp}))$.

Fence Insertion Since our analysis is static and we want to minimize the execution of fence instructions, we avoid placing fences in loops unless absolutely necessary. Parry achieves this by finding cycles in the control flow graph. Then it assigns an edge weight to the cycle edges that is one more than twice the incoming edge weight as illustrated earlier in Figure 2.4. This ensures that even if many orders from outside the loop overlap inside the loop the linear program will prefer edges outside the loop. This heuristic has value if a loop is executed more than once.

Alternate Fence Placements In many cases there are multiple fence placements that are equivalent according to the multi-cut model. That is, there may be edges with weights on similar paths resulting in the same objective function value from the multi-cut linear program. Our implementation selects the edge closest to the source. Our tool is also able to select alternate cuts where necessary and we discuss our experimental evaluation of equivalent alternative fence placements in Section 2.4.7.

2.4 Experimental Results

We have evaluated Parry with four classic concurrent algorithms (Dekker, Lamport, Parker, and Peterson) and three transactional memory algorithms (TL2 [24], TL2 Eager, and TLRW [25]). We downloaded implementations of the classic algorithms from the Musketeer project [4]), TL2 and TL2 Eager are from the STAMP project [64], and TLRW from the Rochester Software Transactional Memory library [61]. TL2 Eager is a variant of TL2. The TLRW implementation is named ByteEager. The three TM algorithms have procedures that are significantly more complex than those of the classic algorithms. For example, TL2 has nearly 400 nodes and more than 250 lines of code in one procedure.

We evaluate all of the algorithms on the x86 and ARMv7 architectures. We chose to work with ARMv7 due to its increasing relevance in all types of computing and its particularly weak

memory model, compared to x86.

2.4.1 Declaration of Execution Orders

We declared execution orders for each of the seven algorithms to benchmark our approach against the author supplied memory fences. We also removed existing fences from the algorithms.

Classic Algorithms Figure 2.6 shows the orders for the four classic algorithms. We got the orders for Dekker from Lesani's dissertation [54] and the orders for Peterson are similar. We got the single order for Parker from a blog post by Dave Dice. Dice wrote that the Parker implementation in the Java Virtual Machine was found to have a bug due to store buffering [26]. We defined an order according to the description of the bug to ensure that the store to the shared variable `_counter` is flushed. We defined the orders for Lamport based on an analysis that we detail in an appendix (TODO which appendix).

TL2 and TL2 Eager Figure 2.10 shows the orders for TL2 and TL2 Eager which we got from Lesani's dissertation [54]. The source code for both algorithms can be found at [22].

RSTM ByteEager Figure 2.13 shows the orders for ByteEager. The source code for RSTM ByteEager algorithm can be found at [66]. The orders stem from the work on TLRW by Dice et. al [25]. They give a detailed account of critical orders which we use here.

During the process of defining orders for each algorithm we also had to find and remove existing fences to prevent duplication. On closer inspection of the code for the store-store order in the `rollback` procedure of ByteEager we were unable to find any mechanism that might enforce the order.

We contacted the original authors to verify our findings. It became clear that they had built the algorithm for architectures where the order was automatically enforced, namely TSO, and they agreed a fence was necessary. We placed a `dmb st` fence after the source of the order to establish a baseline for comparison, noting that an order definition would have made our communication unnecessary.

		x86	ARM7
Dekker	$8 \xrightarrow{W,R} 9$	8:mfence	8:dmb st
	$13 \xrightarrow{W,R} 9$	13:mfence	13:dmb st
	$25 \xrightarrow{W,R} 26$	25:mfence	25:dmb st
	$30 \xrightarrow{W,R} 26$	30:mfence	30:dmb st
Lamport	$8 \xrightarrow{W,R} 9$	8:mfence	8:dmb st
	$14 \xrightarrow{W,R} 15$	14:mfence	14:dmb st
	$31 \xrightarrow{W,R} 32$	31:mfence	31:dmb st
	$37 \xrightarrow{W,R} 38$	37:mfence	37:dmb st
Parker	$44 \xrightarrow{W,*} 46$	44:mfence	44:dmb st
Peterson	$5 \xrightarrow{W,R} 7$	5:mfence	5:dmb st
	$14 \xrightarrow{W,R} 16$	14:mfence	14:dmb st

Figure 2.6: Orders and fences for four classic algorithms

Difficulty The only algorithm without orders already defined in some form was Lamport's mutex. Every other algorithm had research, implementation notes, or existing fences from which orders could be derived. We think this means execution order definition is already implicitly taking place during algorithm design but the information is lost as fence placements during implementation.

Further, in the case of ByteEager's `rollback` procedure, we believe that the speed with which the ByteEager authors were able to diagnose the issue suggests that authors and designers will have relatively little difficulty in defining execution orders during algorithm design.

TL2	LOC	Nodes
TxLoad	75	171
TxStore	121	236
TxCommit	277	398

ByteEager	LOC	Nodes
read_ro	30	64
read_rw	32	73
write_ro	31	93
write_rw	36	122
rollback	25	93

Figure 2.7: Algorithm Procedure Size

2.4.2 Parry's Execution Time

Figures 2.8 and 2.9 show the wall clock time that Parry's top-level `run` procedure takes to insert fences for the TL2 and ByteEager TM algorithms. Figure 2.8 shows results where the exponential order elimination algorithm was used and Figure 2.9 shows results for the linear order elimination algorithm. Notably, the elimination results from the exponential time and linear time order elimination algorithms are identical for all of the evaluated code.

The times were recorded from each stage of Parry's execution, averaged across 100 runs on an Intel Core i5 at 2.4 Ghz with 6GB of RAM with a fully updated version of Ubuntu 14.04 Server.

We include only the TM algorithms here because they are the most complex examples in our evaluation. The size of each procedure for both algorithms in lines of code and control flow graph notes is included in Figure 2.7. The lines of code are recorded without account for inlining except in the case of TL2:`TxCommit` where the majority of the instructions are inlined from a procedure call. The largest case is TL2:`TxCommit` procedure where the control-flow graph has 398 nodes. We have not included TL2 Eager since the size and times were similar to those of regular TL2.

The control-flow graph generation and order elimination account for the majority of the execution time. In the cases where the linear order elimination algorithm is used the graph generation dominates the other parts of our approach. The long execution times for graph generation are caused by a large amount of string manipulation and scanning while working with the LLVM IR in Python. The fence insertion which uses GLPK to run our integer linear program takes little time.

When executing on TL2 for x86 no time is spent on the ILP and a small amount on order elimination. Parry can forgo running the linear program entirely because all of the orders are eliminated. The order elimination only requires a small amount of time because it is immediate for orders where the source and sink instructions exist in an architecture enforced relationship.

2.4.3 Experiments with the Four Classic Algorithms

For the four classic algorithms, Parry inserted the fences shown in Figure 2.6. Notice that each order led to one fence. In each case, the fence is correctly placed and is the best fence possible. We note that Lamport's mutex has two "loops" due to jumps to the start of the algorithm; Parry places a fence directly after the first branch, which is a good choice.

2.4.4 Transactional Memory Algorithms

For the three TM algorithms, Parry inserted the fences shown in Figure 2.10 and Figure 2.13. We have an opportunity to compare those fences with a baseline. The original authors of the transactional memory algorithms inserted fences or fence macros for particular architectures, which we assume are correct for the intended architectures. From those fence placements, the literature, and in some cases consultation with the original authors, we constructed a baseline for each of x86 and ARMv7. Effectively, we acted as an implementer who selects fences; for example, for TL2 we defined an existing fence macro called `MEMBARSTLD` as `mfence` on x86. Similarly, ByteEager uses a memory fence macro `WBR` and the implicit memory barrier defined by the semantics of the `__sync_*` compiler built-ins [1]. One goal of our experiments is to evaluate whether our order definitions and Parry can match the baseline. In the following subsections, we will give a detailed

comparison of both the fence placements and of the resulting performance of the TM algorithms on standard benchmarks.

2.4.5 Impact of Order Elimination

Figure 2.10 shows that for TL2 on x86, Parry eliminated all 7 orders and inserted no fences at all, while on ARMv7, Parry eliminated no orders. Additionally, Figure 2.10 shows that for TL2 Eager on x86, Parry eliminated all 4 orders and inserted no fences at all, while on ARMv7, Parry eliminated one order. Finally, Figure 2.13 shows that for ByteEager on x86, Parry eliminated one order, while on ARMv7, Parry eliminated no orders.

Now let us compare with the baseline. We ask: (1) are there cases where order elimination is necessary to approximate the fence placements from a knowledgeable implementor and (2) can we avoid adding fences altogether using information from the compiler? Our comparison suggests the answer is *yes* in both cases, as we will detail now.

Order elimination prevents the addition of extra fences where the architecture directly enforces an order and an implementer will never insert a fence. For example, load-load orders are automatically enforced on x86. In such cases we can establish a derivation by instantiating an architecture axiom directly and then eliminate the corresponding order. There are also situations like the write procedure for ByteEager where accounting for the `cmpxchg` instruction prevents the insertion of an additional fence.

Additionally, we have exhibited two instances where fences would likely be inserted by an implementer but which actually require no fence. If TL2 is compiled with the `TL2_EAGER` flag, one fence in `TxCommit` can be eliminated on architectures like ARMv7 since the source of the order will not appear in the control flow graph. If TL2 is compiled by Clang on x86, another fence can be eliminated in `TxCommit` due to generated instructions which allow for a transitive order derivation. In these cases a detailed accounting of the control flow graph is important in determining whether an order is already enforced.

2.4.6 Performance Benchmarks

We compare the performance of Parry's output with the baseline using six of the benchmarks from the STAMP benchmark suite v0.9.10 [64] which is designed for testing transactional memory algorithms. We ran the x86 benchmarks on an Intel Core i5 at 2.4 Ghz with 6GB of RAM with a fully updated version of Ubuntu 14.04 Server. The ARMv7 benchmarks were run on an Exynos 5 Dual at 1.7Ghz with 2GB of RAM with the same operating system.

We compiled the results of each benchmark by taking the arithmetic mean of each over 100 runs and then recording the percentage difference between the baseline version and the Parry version.

Importantly everything was compiled with Clang version 3.3. This is the same version used in Parry to generate IR, control flow graphs. Using the same version of Clang to generate the control flow graphs and to compile the algorithms ensures that the assertions we make about the graphs remain valid for the final compiled output.

2.4.7 TL2 and TL2 Eager Measurements

Figure 2.10 shows the fences inserted by Parry alongside the fences placed in the baseline. We associate them by the orders we defined for TL2. For example the last two orders for TL2 correspond with the orders from our running example, `TxCommit`. All of the orders were accounted for by fence macros. This is not surprising given that the authors would have a deep understanding of the algorithm's behavior under weak memory models.

The orders are defined using line numbers which can be referenced in the code which accompanies our project [11]. We have included annotations for the instruction types that should be ordered when they appear in the intermediate representation. For example the order between lines 760 and 1413 in `TxCommit` is a store-store order between the store of a value in the write-set and a store to release the lock for the write-set's address. In this case the line numbers appear to be abnormally distant from one another but, due to procedure inlining, they both appear in the control flow graph for `TxCommit`. We have also included a mapping from the orders in our running example to the line number orders in Figure 2.11.

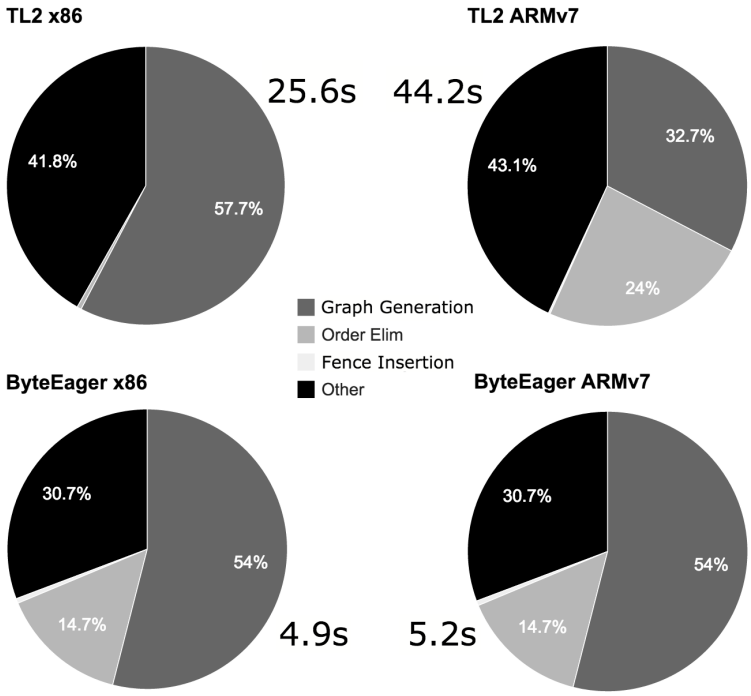


Figure 2.8: Parry Execution Times, Full Order Elimination

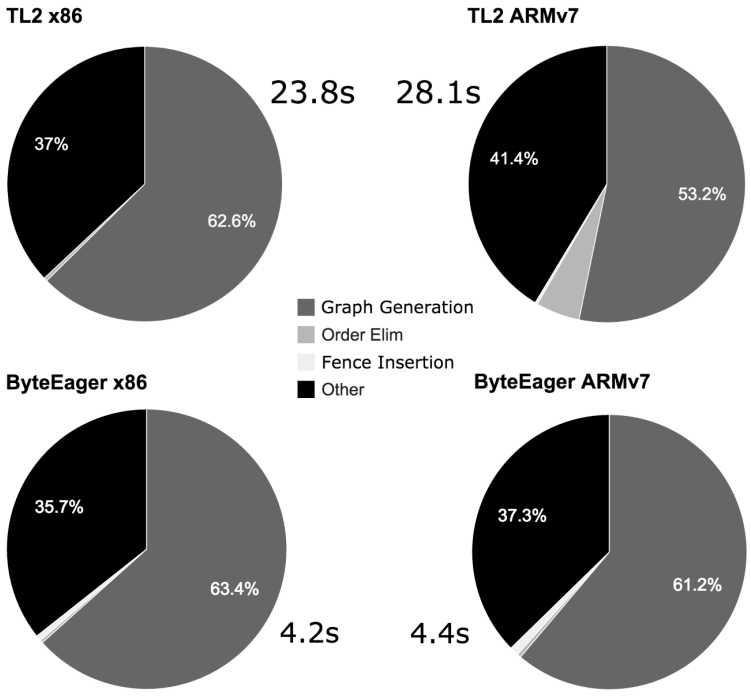


Figure 2.9: Parry Execution Times, Linear Elimination

TL2		**x86**		**ARM7**
`TxLoad`	baseline	ours	baseline	ours
2078 $\xrightarrow{\text{R,R}}$ 2080	—	—	2078:dmb	2078:dmb
2080 $\xrightarrow{\text{R,R}}$ 2082	—	—	2080:dmb	2080:dmb
`TxStore`	baseline	ours	baseline	ours
1886 $\xrightarrow{\text{R,R}}$ 1923	—	—	1920:dmb	1886:dmb
`TxCommit`	baseline	ours	baseline	ours
1555 $\xrightarrow{\text{W,W}}$ 1625	—	—	1555: `ldrex/strex`	1555:`ldrex/strex`
1596 $\xrightarrow{\text{W,W}}$ 1625	—	—	1596: `ldrex/strex`	1596:`ldrex/strex`
760 $\xrightarrow{\text{W,W}}$ 1413	—	—	1669:`dmb st`	1669:`dmb st`
1413 $\xrightarrow{\text{W,R}}$ 1679	1679:`mfence`	—	1679:`dmb st`	1416:`dmb st`

TL2 Eager		**x86**		**ARM7**
`TxLoad`	baseline	ours	baseline	ours
1991 $\xrightarrow{\text{R,R}}$ 1993	—	—	2078:dmb	2078:dmb
1993 $\xrightarrow{\text{R,R}}$ 1995	—	—	2080:dmb	2080:dmb
`TxCommit`	baseline	ours	baseline	ours
760 $\xrightarrow{\text{W,W}}$ 1413	—	—	1669:`dmb st`	—
1413 $\xrightarrow{\text{W,R}}$ 1679	1679:`mfence`	—	1679:`dmb st`	1679:`dmb st`

Figure 2.10: Orders and fences for TL2 and TL2 Eager

	lines	**orders**
`TxCommit`	760 $\xrightarrow{\text{W,W}}$ 1413	W(addr) $\rightarrow$ W(lock)
	1413 $\xrightarrow{\text{W,R}}$ 1679	W(lock) $\rightarrow$ R(x)

Figure 2.11: TL2 Lines to Orders

TL2 x86 The load-load orders in `TxLoad` and `TxStore` and the store-store orders in `TxCommit` should be defined without a fence by an implementer since those instructions can never be seen to move past one another. Parry makes the same determination. As noted earlier we were able to eliminate the final store-load order in `TxCommit` entirely due to Clang's IR output and the transitive order as derived in Figure 2.3. It's unlikely that an implementer would have enough information to make the same determination without the help of our tool.

TL2 ARMv7 For both orders in `TxLoad`, the order in `TxStore`, and for the last two orders in `TxCommit`, Parry inserted the same fence as the authors. Parry also matched the authors for the first two orders in `TxCommit`. They begin with `cmpxchg` instructions which compile to a paired `ldrex/strex` on ARMv7 and require no additional fence.

Note, that the result for the last order in `TxCommit` and the only order in `TxStore` are at different line numbers. Both orders have a path with many edges of the same capacity that can all serve to separate the source from the sink in a minimal cut. The algorithm simply chooses the edge closest to the source in this case. Also, the line number where Parry placed the fence for the second order of `TxCommit` is seemingly before the source of the order. This is due to procedure inlining.

TL2 Eager The eager version of TL2 requires fewer execution orders than the full TL2. The orders in `TxLoad` correspond with the same orders for the full TL2. Otherwise, the difference is the first order in `TxCommit`. As we can see in the example code from Section 2.1, when the algorithm is compiled as eager the source of the first order is removed from the control flow graph by the first `ifndef` in Figure 2.1. This makes the order unnecessary since it does not appear in that procedure's control flow graph.

Performance Let us consider the performance of the TL2 algorithm; the results for TL2 Eager are similar and we omit them here. The performance results in Figure 2.12 for TL2 follow intuition quite closely. In the case of x86 we saw a small improvement since we were able to eliminate a fence in `TxCommit`. The improvement is small because `TxCommit` is called infrequently when

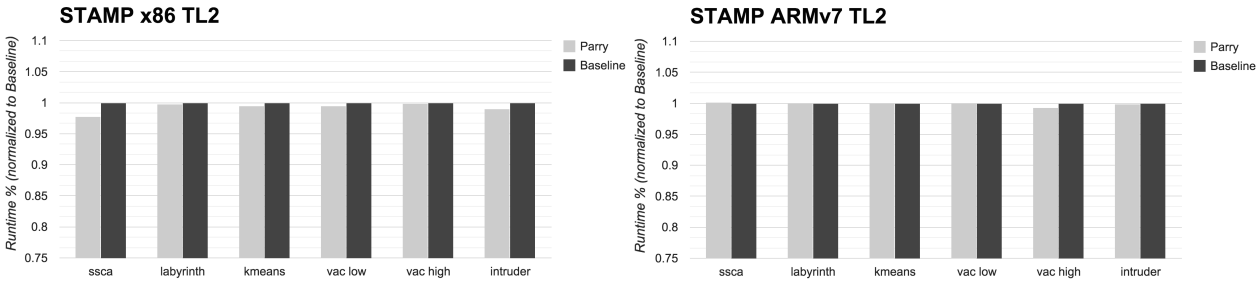


Figure 2.12: TL2 Performance Benchmarks

compared with `TxLoad` and `TxStore` which are the most heavily used procedures in any transactional memory algorithm. Similarly, the results for ARMv7 for the baseline and Parry are nearly identical since the number and placement of fences are nearly identical.

Alternate Fence Placements To test whether different fence placements that are considered equivalent by the multi-cut model might produce real performance differences we considered five alternate fence placements for the TL2 algorithm on ARMv7. We chose this benchmark because TL2 on ARMv7 has the largest number of alternate placements. This is due to the large control flow graphs for the procedures of TL2 and the weaker memory model of ARMv7.

Our results showed that, for every alternate placement, for all six of the STAMP performance benchmark results, the difference between the default fence placement provided by Parry and the alternate was within ±4%. Further only three benchmarks of a total thirty showed greater than a 3% change and two showed greater than 2% change. This suggests that choosing the default is reasonable where performance is concerned.

2.4.8 RSTM ByteEager Measurements

The table in Figure 2.13 contains the fence placements for the RSTM ByteEager implementation along with the memory fences already present in the implementation. Again we note that all orders except the last in `rollback` are accounted for in the implementation of the algorithm. We believe this is due to the author's intimate knowledge of the intended behavior and the detailed account of the fence placements for TLRW in [25].

ByteEager	**x86**		**ARM7**	
`read_ro`	baseline	ours	baseline	ours
125 $\xrightarrow{W,R}$ 128	125:`xchg`	125:`mfence`	125:`ldrex/strex`	125:`dmb st`
`read_rw`	baseline	ours	baseline	ours
163 $\xrightarrow{W,R}$ 165	163:`xchg`	163:`mfence`	163:`ldrex/strex`	163:`dmb st`
`write_ro`	baseline	ours	baseline	ours
186 $\xrightarrow{W,R}$ 196	186:`xchg`	186:`xchg`	186:`ldrex/strex`	186:`ldrex/strex`
`write_rw`	baseline	ours	baseline	ours
228 $\xrightarrow{W,R}$ 238	228:`xchg`	228:`xchg`	228:`ldrex/strex`	228:`ldrex/strex`
`rollback`	baseline	ours	baseline	ours
261 $\xrightarrow{W,W}$ 265	—	—	266:`dmb st`	266:`dmb st`

Figure 2.13: Orders and fences for RSTM ByteEager

```
void set_read_byte(uint32_t id) {
#if defined(BASELINE)
  __sync_lock_test_and_set(&r[id], 1u)
#else
  // NOTE: no WBR macro
  r[id] = 1;
#endif
}
```

Figure 2.14: RSTM

x86 The placements here are noteworthy due to the `xchg` instructions present in the baseline version of `read_ro`, `read_rw`, `write_ro`, and `write_rw`. The source for the orders in the `read_ro` and `read_rw` procedures are calls to another procedure which is not inlined and which uses the `__sync_lock_test_and_set` compiler built-in. When targeting x86 this built-in compiles to the `xchg` instruction which includes an implicit `lock` prefix. This prevents other stores and loads from moving past the `xchg` [17, p. 160]. As a result it can do the same job as the `mfence` that Parry inserts. For our benchmarks we compared the version using the compiler built-in and a modified version which relies on Parry to insert the proper fence as illustrated in Figure **??**. This accurately reflects how an author might rely on an order definition to enforce the correct behavior without the compiler built-in. It also mirrors the definition for SPARC in the original source (not depicted in Figure **??**) which is identical to the version we use to test Parry, except that it includes an explicit `WBR` fence macro.

For `write_ro` and `write_rw` the author's implementation uses the `__sync_bool_compare_and_swap` built-in at the source of the orders. This built-in translates to a `cmpxchg` LLVM IR instruction with a sequential consistency ordering qualifier which in turn compiles to the `xchg` instruction. Parry accounts for the `cmpxchg` by treating it as a fence per the LLVM documentation [72], and consequently it does not add an additional fence to these procedures.

ARMv7 The placements are the same for `read_ro` and `read_rw` as they are for x86 but here the compiler built-in results in a paired `ldrex`/`strex`. Since we have implemented our own simple read, Parry again inserts a qualified `dmb st`. As with x86, when considering the `write_ro` and `write_rw` procedures Parry accounts for the `cmpxchg` instruction and does not add an additional fence.

In the `rollback` procedure a store-store order is required to preserve the expected TLRW behavior when aborting a transaction. We were unable to find anything that would enforce a store-store order in the original implementation. As discussed we contacted the original authors and determined that they did not consider store-store orders when building the algorithm. We placed an appropriate fence for the baseline which Parry matched.

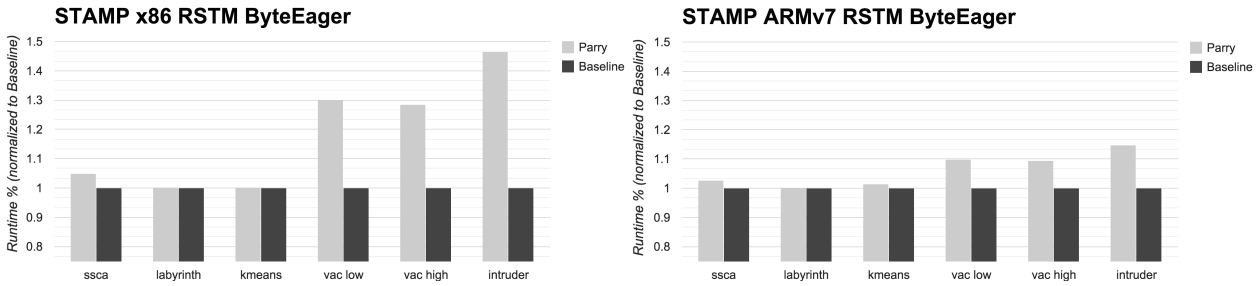


Figure 2.15: RSTM ByteEager Performance Benchmarks

Performance On x86 we expected to see similar results for both Parry and the baseline across benchmarks but in some cases Parry's compiled output was almost 1.5 times slower. Upon further inspection the overhead is due to the modifications we made in Figure **??**. Both the `mfence` and `xchg` instructions can act as an appropriate memory fence for the orders in `read_ro` and `read_rw` and they have similar execution costs to the best of our knowledge. They differ, in that `xchg` also handles the store to memory where our modifications perform the store using a separate instruction.

To test this, we removed the `mfence` from the version which relies on a store and fence. The benchmark results with just a store instruction were comparable to the original using the `xchg`. This suggests that, in these benchmarks, the `xchg` is only marginally more expensive than the store. In future work we could address this by using the `xchg` to enforce orders by replacing the store entirely.

On ARMv7 we see a smaller difference though clearly the same issue exists here: the store and the `dmb st` fence are more expensive than the paired `ldrex`/`strex`.

For both architectures the fence inserted by Parry is only subtly different from the code generated by the compiler built-in but the performance impact is significant. We believe this highlights the importance of finding optimal placements and fences types wherever possible.

2.5 Summary

We have demonstrated that specified orders have a practical implementation using fence insertion that results in fast code. Thus, we can proceed with the task of making reasoning general knowing that when we are done our proofs will result in practical and fast code.

CHAPTER 3

General Reasoning with Specified Orders

Here we will describe how a well chosen memory model can enable general reasoning about lock-free concurrent programs. In particular, we will discuss how the choice of Java's Access Modes with the addition of specified orders supports general reasoning about such programs. Then we will detail our model of Java's Access Modes in full, discuss our work to validate it, relate it to other models empirically, and give an initial metatheory for the model.

3.1 Relating Memory Models

A memory model defines the set of possible executions for a program at the memory access level. When there are more "behaviors" allowed by the model then more executions are allowed. Recall the message passing example execution graph from Figure 1.2 in Section 1.1. If we assume the read, $\mathsf{b} \xrightarrow{\mathtt{rf}} \mathsf{c}$, then under SC there is only one possible execution, namely the total order a, b, c, d. By contrast, on ARM processors the additional behavior of reordering of instructions means there are many more possibilities including the one allowed by SC. In general a weaker memory model admits more behaviors, like access reordering, and therefore more executions. Conversely, a stronger memory model makes more "guarantees" to the programmer and thus admits fewer behaviors and fewer executions.

In Figure 3.1 we have a Hasse diagram [12] of memory models related by the executions they admit. When there is a directed edge between models, the source model admits a superset of the executions of the target model. For example, at the top, x86 admits strictly more executions than SC. As we will discuss later, some of the edges are labeled with the guarantees that the target model provides (i.e. forbidden behaviors in the target).

Recall that sound reasoning about programs must account for all possible executions. Then, ignoring for now [1] the costs of reasoning about a larger set of executions, choosing a memory model further down in the diagram means that demonstrating some property of a program for that model immediately gives the same property for any model above it in the diagram. Thus, for the purposes of general reasoning, we are interested in finding a greatest lower bound in this diagram. If we can perform reasoning for such a model then we can carry that reason up the latice to any stronger memory model by implication.

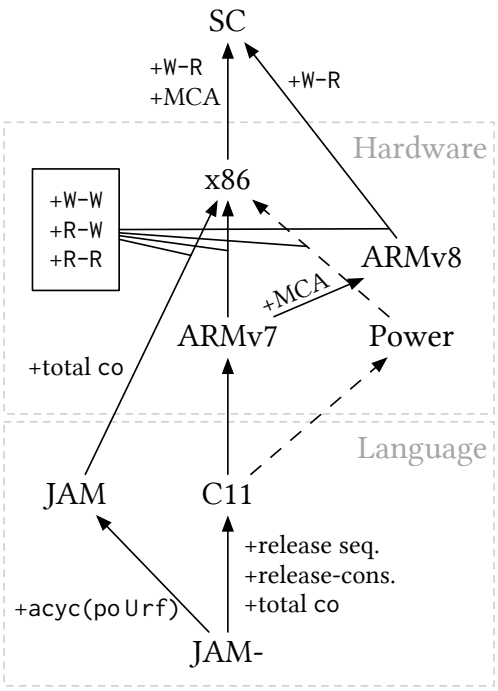


Figure 3.1: Memory Models

We have pinpointed such a bound in the form of a model of Java's new Access Modes [52, 51, 41], hereafter "the JAM", which are similar in spirit to C11's memory orders [15]. The reasons for selecting the JAM are twofold. First, the JAM is designed to be very weak to admit as many compiler optimizations as possible while restricting the particularly pernicious behavior known as *causal cycles* which we will discuss in Section 3.3.1. This restriction is the reason that the JAM is related directly to so few of the memory models in the diagram. By removing the admonition against causal cycles we define the JAM- which admits more executions than any model in the diagram. Second, language memory models are weaker than hardware memory models by necessity because their programs must eventually run on said hardware and advanced compiler optimizations can introduce new behaviors (and therefore more executions). This manifests as the JAM- and C11 appearing lower in the diagram than all of the hardware models.

At this point the reader may wonder, how do specified orders play a roll in facilitating general reasoning? To answer this question, note that specified orders act as a very targeted reduction of

[1] We will address the difficulty of formal reasoning for such a model in Section 4.

the set of possible behaviors for a particular program. Specifically, they rule out executions where the two related memory accesses execute in violation of the specified ordering. Most importantly, specified orders eliminate fewer behaviors than their implementation as fences and this makes them ideal for general reasoning.

For example, placing a `dmb st` fence between accesses `a` and `b` to enforce the specified order in the extended message passing example of Figure 1.3 Section 1.1 results in the ordering of all stores before the fence with all instructions after the fence. In particular it orders `a` and `e` which is unnecessary for correctness.

More generally, this guarantees that a program with specified orders will admit a superset of the executions for the same program on the same model with the inserted fences. Since the JAM- admits strictly more executions than the other models in the diagram regardless of specified orders it must admit more executions with specified orders than the compiled output with fences. Taken together this means that reasoning performed for the JAM- with specified orders is sound for any of the stronger memory models in our diagram.

3.2 We're JAMing

Developers can make use of the JAM, through the `VarHandle` API included in the JDK version 9. There are four access modes: plain, opaque, release-acquire, and volatile. Regular reads and writes of shared variables are considered plain mode, and the `VarHandle` API allows reads and writes to be annotated with one of the other three modes. The specified intent of the JAM is that each mode provides progressively more guarantees about the behavior of its accesses and admits fewer possible executions at the expense of some performance.

$$\text{plain} \sqsubseteq \text{opaque} \sqsubseteq \text{release-aquire} \sqsubseteq \text{volatile}$$

Plain mode gives virtually no guarantees and the compiler is allowed free reign in optimizing such memory accesses which results many new weak memory behaviors. On the other hand volatile mode provides SC semantics when it is used for all accesses, but requires a memory barrier

for each access which can make execution slow [2]. These modes are inspired by the memory orders of C11 [15] but there are significant differences in behavior that we detail in Section 3.3.

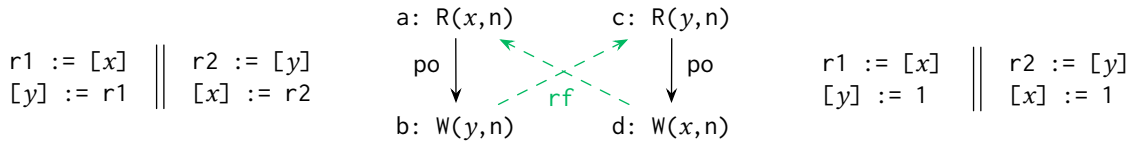
In Section 3.4 we present an axiomatic model that includes all of the access mode, fences, and atomic reads-writes of the JAM which we have constructed using the `cat` language. `cat` allows us to leverage the Herd tool [5] to give definitive answers to questions about example programs, called litmus tests. Litmus tests are designed to highlight specific behaviors in memory models. Herd enumerates all possible executions for a litmus test and determines which are allowed according to the model. Then, if at least one execution is allowed, the behavior illustrated by the litmust test is allowed by the model.

In Section 3.5 we use Herd with more than 80 litmus tests drawn from prior research to help validate our model by comparing it to ARMv8, C11 and x86. Our goal for these comparisons is to show that there are no unexpected differences in behavior and to further define the relationships between the compared models. For example, the conventional wisdom is that language memory models will exhibit more behaviors than architecture memory models due to aggressive compiler optimizations. Thus, Java's access modes should permit more behaviors than ARMv8. If that is not the case then it should be due to a deliberate design decision and not a bug in the definitions. For all 80 litmus tests our model behaves according to our expectations and the documented design of the access modes.

In Section 3.6 we give an alternate instatiation of our model in Coq and we prove three key theorems: absence of causal cycles when all reads are "release" reads, sequentially consistent semantics under proper synchronization (DRF), and a guarantee that each stronger mode admits fewer executions [85]. These theorems further validate the definitions of our model, give more evidence that the model is complete with respect to the documentation, and clearly demonstrate that the semantics is suitable for formal reasoning.

Finally, the partial order on same-location writes in the JAM specification represents a significant departure from the conventions of existing memory models. In Section 3.7 We show that the impact of switching to a partial order is "unobservable" in any example program executed

[2] This property, of progressively greater guarantees, is called monotonicity and we prove it as a theorem for our model in Section 3.6



(a) Out of Thin-air (b) Causal Cycle (c) Load-buffering

Figure 3.2: Causal Cycles

using our model and thereby show that the JAM can adopt a total order on writes to the same location. The simplicity of our model shines through in our proof, which makes it clear that our reasoning is not applicable to the more complex axiomatic models of RC11 and ARMv8.

3.3 Distinguishing Features of the JAM

While the JAM was inspired by the access modes of C11, it makes several departures from C11 and other memory models that are worthy of consideration and a challenge for formalization. First, it contains a broad and simple definition of causal cycles for the purposes of ruling out so called thin-air reads [9]. Second, it sheds legacy features like the release sequences and consume-reads of C11. Finally it includes a non-total ordering on writes to a particular memory address, which is called the coherence order. Here we will discuss how each of these differences will impact any formalization effort for the JAM.

3.3.1 Acyclic Causality

In Figure 3.2a we have a classic example program which can exhibit a so called "thin-air" read under sufficiently weak memory models. The question is, at the end of execution can $x = y = 42$? Intuitively, assuming both x and y are initialized to 0, this program can't generate 42 "out of thin-air", but many axiomatic models do not exclude such executions from the set of all candidate executions.

Observe that, in any execution that allows $x = y$, there must be a cycle in the program order and reads-from relations, as illustrated in Figure 3.2b. The JAM explicitly forbids such cycles, but at the cost of forbidding some behaviors which may be beneficial for performance.

For example, consider the classic load-buffering (LB) litmus test in Figure 3.2c, where the question is, can `r1 = r2 = 1`? If performance was the sole concern in the design of the JAM this behavior would be allowed because the reads can be reordered with the writes with the aim of improving performance. Unfortunately, this example also exhibits the same cycle in the program order and reads-from, so it is forbidden by the JAM.

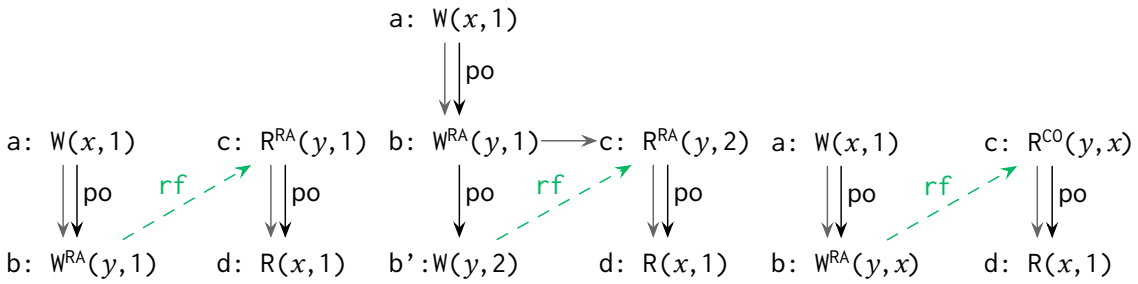
The problem of differentiating these kinds of examples has been studied at great length by memory model researchers. Another approach is to forbid cycles in `rf` and a subset of program order based on a notion of dependency. Sadly, this too has very subtle issues, as outlined by [9]. The original, formal Java Memory Model [60] attempted to address the issue of causal cycles in its full generality. More recently the Promising semantics of [43] introduced a novel "promise" mechanism to model compiler optimizations for this purpose.

In all of these cases the complexity of the resulting models makes them hard to understand and hard to test. Instead, the JAM specification adopts a simple solution by forbidding cycles in the program order and reads-from. While this does forbid the behavior of the second example at the cost of some performance [68], it gives us yet another opportunity to build a simpler model.

Where the diagram of Figure 3.1 is concerned, this approach to forbidding causal cycles places the JAM in a position where it is incomparable to many other memory models in terms of the executions it allows. This is the motivation for JAM-. As we will see in the comparisons of Section 3.5, if we remove the acyclic causality requirement from the JAM the resulting model, JAM-, admits strictly more executions than the other models in the diagram. The mechanism by which the JAM forbids causal cycles, $\texttt{acyc}(\texttt{po} \cup \texttt{rf})$, which we discuss in 3.4, appears as an added guarantee when moving from the JAM- to the JAM up the diagram.

3.3.2 Letting Go of Release Sequences

Release sequences and consume-reads can be seen as specialized variants of the release-acquire memory order in C11 and release-acquire mode in the JAM. The idea is that, in certain cases, it's possible to get the same guarantees as a release-write and acquire-read pair, like the one Figure 3.3a, with less synchronization. The result is faster execution in some contexts.



(a) Message Passing, Fixed (b) Release-sequence (c) Consume-read

Figure 3.3: Release sequences and Consume-reads

Release sequences can be used when a write that is after a release-write is read by an acquire-read. Consider the message passing variant in 3.3b where b is followed by another distinct write to y, b'. If c reads from b' then the release sequences of C11 would guarantee that d would see a through the orderings depicted with gray edges. Then d will read the value 1, just as it would if c read from b. Without the guarantees of release sequences, b' must also be a release-write to get the desired outcome.

Consume-reads are used with a release-write when the memory accesses which must be ordered after the consume-read are data dependent on the value of the read. Consider the message passing variant in 3.3c where c is annotated with CO and reads a pointer that determines the memory address that d reads from. Some architectures enforce the ordering of c and d in the presence of such data dependencies without the extra synchronization that would result from making c an acquire-read. This can speed up execution in those settings.

The JAM does not include the guarantees of release sequences or any way to annotate a read as a consume-read. This is a design choice in favor of simplicity in the model and it gives us the opportunity to build a more readable and more easily testable model.

Additionally, this means that the JAM admits behaviors that C11 explicitly forbids. In the case of release sequences, the JAM does not guarantee that reads from writes after a release will result in synchronization. In the case of consume reads there is simply no way to use data dependencies to forbid reorderings. This is one reason that the JAM- appears bellow C11 in the diagram of Figure 3.1. Both features appear as added guarantees in the label for the edge from

JAM- to C11.

3.3.3 Partial Coherence Order

The JAM specification makes no provision for a total ordering of writes to a given memory location which is a standard feature in other memory models. The consequences of this design choice manifest in subtle ways.

For example, the standard definition of atomicity for read-writes relies on a total coherence order. Borrowing the definition from [85], for a read-write, RW, the write it reads-from, W_1, and the coherence order, $\xrightarrow{co}$, we have:

$$W_1 \xrightarrow{rf} RW \implies \neg\exists W_2, W_1 \xrightarrow{co} W_2 \xrightarrow{co} RW$$

Taken together with a total coherence order this means that the atomic pairs of writes and read-writes are always totally ordered. Since the JAM makes no such guarantees regarding normal writes there are ambiguities like the one in Figure 3.4. There, the write-read-write pairs are not ordered across threads since there is no global relationship between writes.

As a consequence, our model must include extra contraints for read-writes while being careful not to over-constrain normal writes which would otherwise be concurrent. We detail our approach in Section 3.4.5.

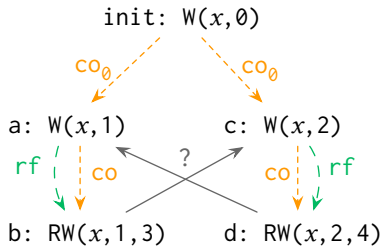


Figure 3.4: Concurrent Read-writes

The lack of a total coherence order is the final differentiating feature between the JAM- and C11 in our diagram of memory models. Similar to release sequences and the release-consume memory order it appears as an additional guarantee when moving from the JAM- to C11 in Figure 3.1.

Separately, our drive for simplicity in the definitions of the model has yielded a key insight where the coherence order is concerned. While we have remained faithful to the documentation and personal communications with the authors in modeling a partial coherence order, we will show that the effects of that choice are not observable under our model. Specifically, we show that it is impossible to construct a litmus test that behaves differently in the presence or absence

of a total coherence order.

3.4 Axiomatic Model

The JAM has six components: plain mode, opaque mode, release-acquire mode, volatile mode, fences, and atomic read-writes. Each of the modes, from plain to volatile, provides strictly more guarantees than the previous mode.

To model the JAM we have constructed an axiomatic semantics. The definitions of our model act as a predicate over candidate executions which include memory events, e.g. reads and writes, and relations over those events, e.g. reads-from and program order, in the style of [5].

Our definitions focus on two key concepts. The first is an acyclicity requirement for the coherence order. Aside from the restriction of causal cycles, this is the only mechanism by which executions are forbidden. Thus, every unwanted execution must exhibit a cycle in the coherence order. The second is an intuitive notion of *visibility*, which represents when one memory access has "seen" the effects of another memory access. The behavior of the three modes and fences are modeled as small extensions to this relationship.

In each of the following subsections we detail the extensions to our model for each component. The full model can be viewed in Appendix C.

3.4.1 Plain & Opaque Mode

In the JAM documentation, plain mode accesses are given virtually no guarantees when they occur in different threads without correct synchronization. Opaque mode, on the other hand, does provide some cross thread guarantees which form the basis for the rest of the memory model. There are some subtleties involved in the documented relationship between plain and opaque mode accesses so we will address them together. First, the main properties that opaque mode accesses guarantee are:

Bitwise Atomicity Reads will see the value of only one write. Opaque mode guarantees that reads will not see mixed bits from different writes.

Write Availability Writes can be read by later reads. The intent is to avoid a situation (e.g. in a spin-loop) where repeated reads never see a write in another thread because they are optimized by the compiler to execute only one time. When an optimization like this happens, the availability of the write for the read depends on when the read is executed [16].

Acyclic Causality A read should not influence its own value. As described in Section 3.3.1, this forbids counter-intuitive behavior like thin-air reads.

Coherence The order of writes should respect visibility and should agree with the way that reads observe their order. For example, one guarantee (of four we will define later) is that a read should be paired with the last write that it knows about and not an earlier one.

Our model of opaque mode focuses on acyclic causality and coherence. In keeping with other axiomatic models built for Herd [74, 47, 3], our model pairs each read with a single write and there is no accounting for optimizing reads out of loops as Herd does not support them.

Plain Coherence. The JAM documentation does not include a coherence guarantee for plain mode accesses. This follows the approach in the C11 documentation but it departs from formal models for C11.

To see why plain mode accesses should be included in the coherence ordering guarantees, note that plain mode accesses are safe to use within a critical section guarded by a lock according to the documentation. The idea is that code which is properly synchronized with locks will have single threaded semantics for the duration of the critical section. Thus, the model shouldn't require accesses to be annotated with anything stronger than plain mode.

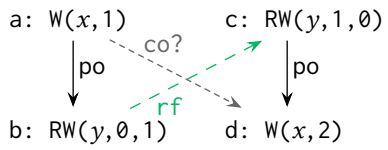


Figure 3.5: Message Passing Coherence

Then, consider the message passing variant in Figure 3.5. Here, the purpose of this pattern is to signal the end of the critical section through atomic read-writes that unlock, b, and lock, c, the variable y. For any lock to work correctly, the accesses which are program order before

```
let vo = rf+ | po-loc

let wwco(rel) = rel & ~id
              & loc & (W * W)

let coww = wwco(vo)
let cowr = wwco(vo;invrf)
let corw = wwco(vo;po-loc)
let corr = wwco(rf;po-loc;invrf)

let coinit = wwco((IW * W))
let cofw = wwco((W * FW))
let co = coww | cowr | corw
       | corr | cofw | coinit

acyclic co

let opq = O | RA | V
acyclic (po | rf) & opq
```

Figure 3.6: Opaque Mode

the unlock, a $\xrightarrow{\text{po}}$ b, must be visible to accesses after the lock, c $\xrightarrow{\text{po}}$ d (for now we leave the mechanism that enforces these orderings unspecified). In particular the effects of the write a should be visibile to the write d, a $\xrightarrow{\text{co?}}$ d.

However, even if the unlock and lock enforce program order of the plain writes, and thereby show that a is visible to d, we would not be able to derive a $\xrightarrow{\text{co}}$ d because the coherence rules do not apply to plain accesses. This stands in contrast to formal models of C11 [47] where the coherence order and happens-before relations apply to plain accesses. As a result, our model extends the coherence order guarantees to plain mode accesses and only the extra guarantee of acyclic causality is left to opaque mode.

Herd Model. Figure 3.6 details our axiomatic model of the JAM's plain and opaque modes as defined in the `cat` modeling language. `cat` includes the following built in operations: `|`, `&`, `;`, `+`, `~` are relational union, intersection, composition, transitive closure, and complement. New constructs are defined with `let`. Filters, like `wwco`, are defined on relations using `let F(R) = ...` and applied with `F(R)`. Finally, models can include checks like `acyclic` for relations.

Herd provides several built-in, ambient sets and relations for models defined in `cat`. `IW`, `W`, `FW`, `R`, `O`, `RA`, `V` are the sets of initial writes for each location, plain mode writes, the final plain mode writes to each location, plain mode reads, opaque accesses, release-acquire accesses, and volatile accesses. `po`, `po-loc`, `rf`, and `invf` represent the program order, program order per-location,

reads-from, and inverted reads-from relations. `loc` relates memory accesses to the same location and `id` is the identity relation. Note, that we do not use Herd's built-in coherence relation but rather define our own without a total ordering.

We define visibility order, `vo`, using two properties, `po-loc` and `rf+`. In the first case, given two memory accesses to the same location in the same thread the second should see the effects of the first. In the second case, the sequence of one or more reads guarantees that the final read is executing in a context where the effects of the write are visible. This definition for `vo` guarantees only the most basic forms of ordering which is important given the inclusion of plain mode accesses.

Then, to satisfy the coherence requirement we first ensure that the coherence order includes only distinct writes to the same location, `wwco`, and adopt the four behaviors originally identified for C11 by [8]:

`coww` Two such writes ordered by visibility are similarly coherence ordered.

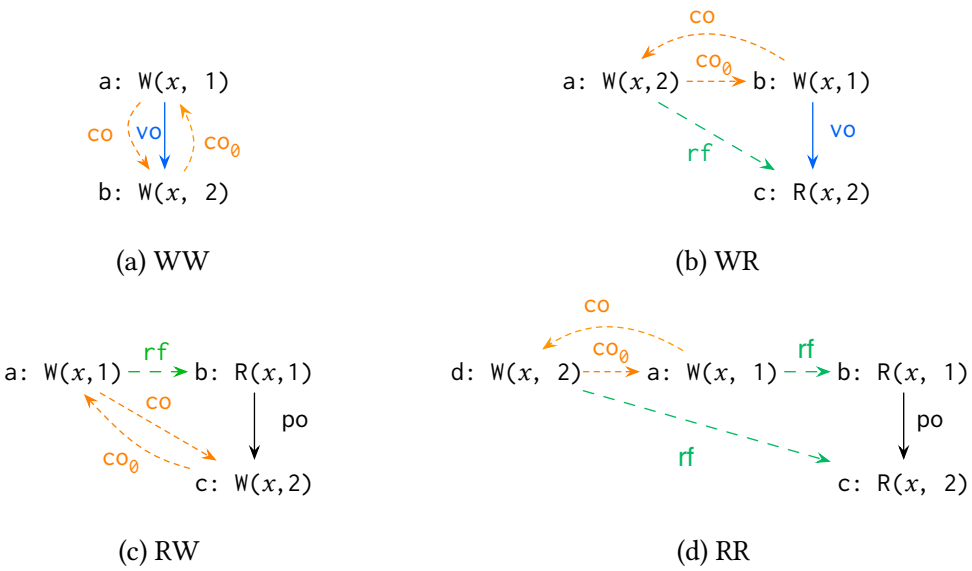
`cowr` A read chooses the last write it has seen.

`corw` A write that follows a read of the same location in program order is coherence ordered after the write paired with the read.

`corr` Program ordered reads to the same location order their writes in the same way.

We also order the initialization write before all writes of the same location, `coinit`, and we order all writes before the final write of the same location, `cofw`. In the rest of the paper these relationships will be distinguished as $\texttt{co}_0$ for clarity but they are treated the same as any other `co` edge by the model.

These six rules make up the definition of the coherence order, `co`, until we discuss read-writes. Then, the primary mechanism by which the model forbids executions is through requiring the coherence order to be acyclic, `acyclic co`. In Figure 3.7, we give examples of how each coherence rule forbids an execution by showing a cycle in the coherence order. In every example, the final write of 1 is assumed to be coherence order after all other writes, $\texttt{co}_0$.



(a) WW (b) WR

(c) RW (d) RR

Figure 3.7: Coherence

In Figure 3.7a, if the write, a, is visible write, b, to the same location then a is not the last write. In Figure 3.7b, if the read c has seen the last write b then it should not be able to read an older write a. In Figure 3.7c, if we can follow the read of a to b to a later write, c, then a can't be the last write. In Figure 3.7d, given two reads in program order, b and c, if b has seen the last write then c should not be able to read an older write.

Finally, we must forbid causal cycles for opaque mode access. We define opq to be any opaque or stronger access since any guarantee provided by opaque should apply to stronger access modes. Then, we forbid cycles in the union of program order and reads-from, qualified for opaque mode accesses: `acyclic (po | rf) & opq`.

As we will see in the following sections, these base definitions allow us to model nearly every other component by extending vo. The lone exception is atomic read-writes, for which we extend the coherence order directly.

3.4.2 Fences

The JAM supports five types of fences: release, acquire, load-load, store-store, and full. Programs often include fences to enforce the order of memory access effects before and after the fence. We update vo by extending visibility to rfso+ and abstracting over these fence types with specified orders.

```
with to from linearisations(M\IW, cofw | rf | into)
let spushto = to+ & (domain(spush) * domain(spush))
let rfso    = rf | svo | spush | spushto;spush
let vo      = rfso+ | po-loc
```

Note that trace order, `to`, is a total ordering of all memory accesses (except for initial writes) as constructed by Herd's built-in `linearisations` function. Trace order respects `cofw`, `rf`, and all intra-thread orderings, `into`, induced by specified visibility orders, push orders and later release-acquire memory accesses and volatile memory accesses.

Specified Visibility Orders. To see how specified visibility orders create intra-thread visibility between accesses, recall the event graph for the message passing example from Section 1.1, reproduced here in in Figure 3.8.

By specifying that a $\xrightarrow{\texttt{svo}}$ b and c $\xrightarrow{\texttt{svo}}$ d we can show that d could not have read 0 from the initial write to x. First, we assume that the initial write to x is coherence order before a, init $\xrightarrow{\texttt{co}_0}$ a. Then, assume that d has read from the initialization, R(x, 0). We will show a contradiction. In the graph we have that, a $\xrightarrow{\texttt{svo}}$ b $\xrightarrow{\texttt{rf}}$ c $\xrightarrow{\texttt{svo}}$ d. By the definition of `rfso` we have the three edges that combine to show a $\xrightarrow{\texttt{vo}}$ d. Then, by `cowr` (Fig. 3.7b) we have that a $\xrightarrow{\texttt{co}}$ `init` which is a cycle in co and a contradiction.

Specified Push Orders. Specified push orders create visibility relationships in two ways. The first is an intra-thread visibility ordering between the two push ordered instructions like svo. This appears as the spush in the definition of `rfso`.

The second, is a cross-thread ordering that emulates the standard total ordering of two full fences. Given two push orders, the head of the first will be visible to tail of the second, or vice-versa based on the current ordering of the heads in `to`. The ordering of the heads is recorded as `spushto` which we connect with the additional visibility ordering of one tail using, `spushto;spush`.

To see how push orders emulate full fences, consider the store-buffering litmus test in Figure 3.9. The question is, can both reads take their values from the initialization?

First, we assume the gray edge between the two `fence` instructions, `fence` $\longrightarrow$ `fence`, which represents one side of the total ordering provided by full fences. Then, we assume that d reads

from the initial write to x and show a contradiction. Since fences also provide the intra-thread orderings a $\longrightarrow$ fence and fence $\longrightarrow$ d, we can see that a is ordered before d. Then by cowr it must be that a $\xrightarrow{\text{co}}$ init, which creates a cycle with init $\xrightarrow{\text{co}_0}$ a and a contradiction. On the other side of the ordering between fences a similar argument applies if b reads from initial write to y. Thus, b and c could not have read from the initial writes to x and y in the same execution.

Push orders are more direct but capture the same ordering. In Figure 3.10 both write-read pairs are push ordered.

First we assume one side of the total trace order between the two writes, a $\xrightarrow{\text{to}}$ c. Again, we assume that d reads from the initial write to x and show a contradiction. By spushto;spush, we have a $\xrightarrow{\text{vo}}$ d which means that d has seen a and the same reasoning with cowr that we used with the fences applies.

Note that including spushto in vo in the place of spushto;spush would give the desired visibility between the heads and tails of two orders by transitivity, but it would also give an ordering to the heads of the push orders which does not otherwise exist.

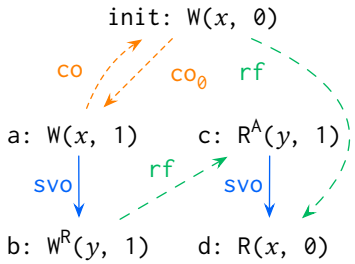


Figure 3.8: Specified Visibility

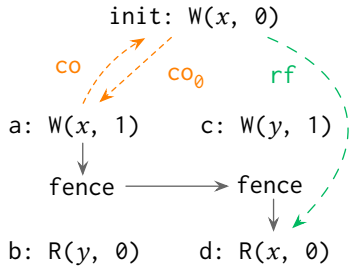


Figure 3.9: Fences, One side

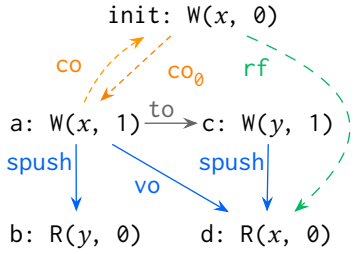


Figure 3.10: Push order, One side

3.4.3 Release-Acquire Mode

We can now make small extensions to incorporate other access modes into our model. First, we define release-acquire mode following the standard set by other models like C11 [15] and ARMv8 [74].

```
let rel = W & (RA | V)
let acq = R & (RA | V)
let ra  = po;[rel] | [acq];po | rfso
```

```
let vo = ra+ | po-loc
```

We define release writes, `rel`, to be any write marked as release or volatile. We define acquire reads, `acq` to be any read marked as acquire or volatile. Again, any guarantee provided by release-acquire mode should hold for volatile mode.

We update the `vo` definition from opaque mode with fences by extending what was previously `rfso+` to be `ra`. We add edges from memory accesses for any location to a release write that is later in program order, `po;[rel]`. We also add edges from an acquire read to program order later opaque memory accesses for any location, `[acq];po`. Note that the documentation makes clear that plain accesses should be ordered by release writes and acquire reads so long as the types are bitwise atomic.

To see, how the release-aquire extension to opaque mode works, consider the message passing example from the Section 1.1. Note that, we use superscript `RA` for release writes, W^{RA}, and acquire reads, R^{RA}. Later we will use `V` for volatile accesses. Then, if we adopt a release write for `b` and an acquire read for `c` we can show that, `a` $\xrightarrow{vo}$ `d`, and the reasoning is the same as for specified visibility orders in Section 3.4.2.

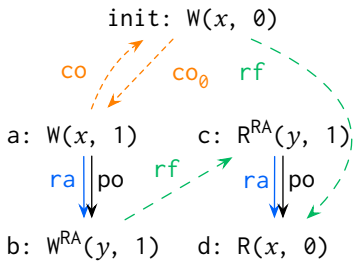


Figure 3.11: Release-Acquire MP

We note that our definitions suggest that, if all reads were release reads, we could derive a visibility relationship between a read and itself in executions exhibiting causal cycles. In section 3.6, we prove that visibility cycles are a contradiction in our model and, as such, causal cycles are forbidden when all reads are release-reads.

3.4.4 Volatile Mode

Volatile mode is a further extension of the visibility order in release-acquire mode. We update `vo` by extending `ra` to `vvo`. Here `volint` creates edges from any access to a volatile read that is later in program order, `po;[V & R]`, and from an volatile write to any program order later opaque

memory access, `[V & W];po`.

```
let vol = V
let volint = po;[vol & R] | [vol & W];po
let push = spush | volint
let vvo = ra | pushto;push | push
let vo = vvo+ | po-loc
```

These new edges preserve program order for any access before or after the volatile access when combined with the visibility definition of release-acquire. We also extend `spush` with `volint` edges so that we can leverage the same visibility relationships induced by push orders for volatile accesses.

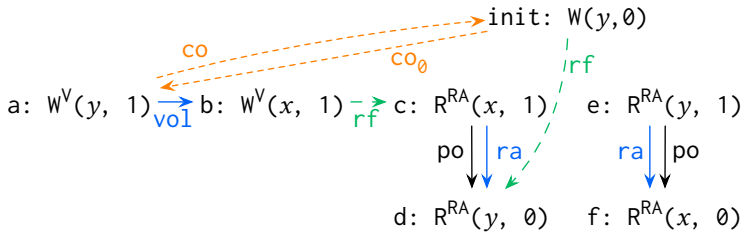
A simpler approach would be to translate the total trace ordering into visbility edges. That is, we could replace the definition of `vol` above with the following:

```
let vol = ra | spush | volint | to & (V * V)
```

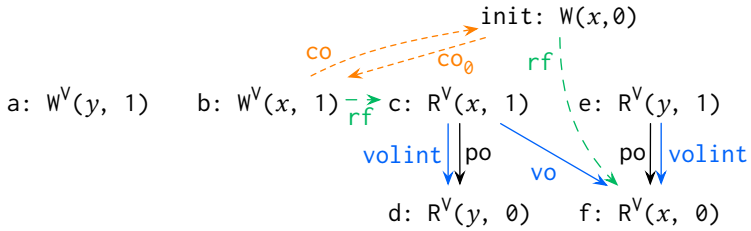
This approach more directly encodes the cross thread ordering guarantee of sequential consistency, but, unfortunately, this is too strong.

Consider a modified version of the classic IRIW litmus test in Figure 3.12a. The question is, can both `d` and `f` read from the initial write to x and y? As outlined in the JAM documentation the acquire reads in this example are allowed to see the writes in different orders, so both can read from the initial writes for x and y. However, if volatile mode accesses are totally ordered as in the proposed definition, then we have either a $\xrightarrow{\texttt{vo}}$ b or b $\xrightarrow{\texttt{vo}}$ a. In Figure 3.12a we have the first case. Then we have a $\xrightarrow{\texttt{vo}}$ d+ because a $\xrightarrow{\texttt{vol}}$ b $\xrightarrow{\texttt{rf}}$ c $\xrightarrow{\texttt{ra}}$ d. Then by `cowr`, we have that a $\xrightarrow{\texttt{co}}$ `init`, which is a contradiction. In the other case, b $\xrightarrow{\texttt{vo}}$ a, we have a contradiction when `f` reads from the initial write to y.

Instead, we extend our notion of push orders and define `push` to include both `spush` and `volint` edges. This has the same effect as push ordering accesses related by `volint`. As we will demonstrate later, using a matching litmus tests, this guarantees the correct behavior for the release-acquire variant of IRIW because it enforces no direct ordering between the two writes.



(a) IRIW Release-Acquire



(b) IRIW Volatile

Figure 3.12: IRIW Variants

Importantly, it also gives the correct behavior when the reads are volatile, which should be SC semantics. Consider Figure 3.12b which has one of the two possible orderings given by `pushto;push` when the reads are volatile. This correctly establishes $\texttt{c} \xrightarrow{\texttt{vo}} \texttt{f}$ creating a contradiction for the opposite thread's final read. As before the other direction of the total order forbids the other read.

3.4.5 Atomic Read-Writes

The behavior of atomic read-writes is the only part of the JAM that is not modeled by extending vo. Recall Figure 3.4 from Section 3.3. We must take care to ensure that we do not allow concurrent read-writes in the presence of a non-total coherence order. To achieve this we update the coherence order co with two additional rules:

```
let corwexcl = wwco((rf;[RW])^-1;co')
let corwtotal = wwco(((RW * W) | (W * RW)) & to)
let rec co = ... | crwexcl | corwtotal
```

The first, `corwexcl` ensures exclusivity in the relationship between the read-write and its paired write. Note that, for clarity, we separate out `corwexcl` even though it recursively refers

to co in its definition. As illustrated in Figure 3.13, if there is a write co-after the one paired with the read-write, a $\xrightarrow{co}$ b then it is co-after the read-write, c $\xrightarrow{co}$ b.

Recalling the example of Figure 3.4, this exclusivity is not enough to prevent concurrent read-read-write pairs to the same location. Since exclusivity uses the ordering of other writes with the paired write we could consider a total ordering just for paired writes.

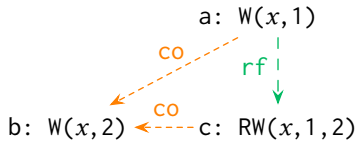


Figure 3.13: Read-write Exclusivity

```
let pairedw = domain(rf;[RW])
let corwtotal = wwco((pairedw * pairedw) & to)
```

This suffices to forbid concurrent read-write chains to the same location. In Figure 3.4, by `corwtotal` we have a $\xrightarrow{co}$ c or c $\xrightarrow{co}$ a. Then, in the first case, we have b $\xrightarrow{co}$ c by `corwexcl`. The other side is similar. However, this approach inadvertently orders writes that could be concurrent under a partial coherence order. Instead, as in the first definition, we choose a total ordering with the read-write itself.

Now any chain of read-writes will be exclusive without unnecesesarily ordering regular writes. We will show that in Figure 3.4 either, b $\xrightarrow{co}$ c or d $\xrightarrow{co}$ a. By `corwtotal` we have d $\xrightarrow{co}$ a or a $\xrightarrow{co}$ d. In the first case we are done. In the second case, we consult `corwtotal` again and we have b $\xrightarrow{co}$ c or c $\xrightarrow{co}$ b. In the first case we are done. In the second case, by assumption we have a $\xrightarrow{co}$ d and c $\xrightarrow{co}$ b. Then we can apply `corwexcl` to both to derive c $\xrightarrow{co}$ d and d $\xrightarrow{co}$ c, which is cycle and a contradiction. Importantly this reasoning can be applied repeatedly to any chain of read writes to achieve exclusivity.

We note here, that JAM makes no intra-thread ordering guarantees for atomic read-writes even though they exist on some architectures like x86 [69]. We discuss this in more detail in Appendix D.

3.4.6 Summary

Our axiomatic model of the JAM is complete, covering all four modes, atomic read-writes, and all five fence types in less than 40 lines of definitions. Moreover each component is implemented as a modest extension to just two relations which makes it readable. Now we must demonstrate that the model is consistent with expectations about the behavior of the JAM as outlined in the documentation. We discuss the results of our comparison with ARMv8, RC11 and x86 in the next section.

3.5 Validation

In this section we validate our formalization of the JAM by comparing litmus test outcomes for the JAM with the outcomes for ARMv8 [74], RC11 [47] and the model for x86 included with Herd. We use these comparisons to show that there are no unexpected differences in behavior between each pair of models. For example, we expect the JAM to permit more behaviors than ARMv8 with the exception of litmus tests like load-buffering (Fig. 3.2c) because the behavior is forbidden by the JAM's definition of causal cycles. Additionally, we use the test results to establish an empirical relationship between the JAM and the other the memory models in the diagram of Figure 3.1.

We run the tests using the Herd tool [5]. Herd exhaustively enumerates all possible executions of a litmus test and checks if each execution is allowed by the model being tested. We can then compare the three possible results for each model to see how often executions are allowed: `Always`, `Sometimes`, and `Never`. In terms of the behavior being tested `Always` and `Sometimes` mean that the behavior is allowed, while `Never` means that it is not allowed. Weaker memory models should allow more behaviors and see more `Always` and `Sometimes` results, while stronger memory models should allow fewer behaviors and see more `Never` results.

Our test suite is built with litmus tests taken from existing research for the three models we compare against. Importantly we did not modify any of these tests. However, Herd uses different built-in relations to refer to the access types of each model. For example, Herd provides the `L` built-in to refer to release-writes in ARMv8 litmus tests and the `REL` built-in to refer to release-

[74]

name	ARMv8	JAM
WRC+addrs	Never	Never
LB+data+data-wsi	Never	Never
W+RR	Never	Never
totalco	Never	Never
PPOCA	Sometimes	Sometimes
IRIW	Sometimes	Sometimes
IRIW+addrs	Never	Sometimes
IRIW+poaas+LL	Never	Sometimes
IRIW+poaps+LL	Never	Sometimes
MP+dmb.sy+addr-ws-rf-addr	Never	Sometimes
WW+RR+WW+RR+wsilp+poaa+wsilp+poaa	Never	Sometimes
LB	**Sometimes**	**Never**

Figure 3.14: ARMv8 Litmus Test Comparison

writes in C11 litmus tests. Thus, for each comparison we include a mapping between the access types of the other model and the access modes of the JAM.

For each comparison we define the mapping between the built-in relations of the two models, we detail our expectations for the results, and then discuss the results of the comparison. As we will see our model behaves as expected in all cases.

3.5.1 Comparison with ARMv8

Herd provides several built-in relations for the ARMv8 litmus tests: `M` for normally memory operations, `L` for release writes, `A` for acquire reads, and `DMB.SY` for full fences. There are no SC/Volatile accesses.

Mapping. We map every memory access to opaque mode with `opq = M`. This is conservative with respect to our expectation that the JAM permits more behaviors than ARMv8. If we were to map some accesses to plain mode they would be allowed to exhibit causal cycles. Thus, mapping

all memory accesses to opaque mode means the JAM model will permit fewer behaviors.

We map release-writes to release-acquire mode writes, `rel = L` and acquire-reads to release-acquire mode reads, `acq = A`. Note, that `M` includes `L` and `A` so release writes and acquire reads will also be treated as opaque mode accesses. Finally we treat all accesses with a full fence between them in program order having a specified push order.

```
let opq = M
let rel = L
let acq = A
let spush = po;[DMB.SY];po
```

Figure 3.15: ARMv8 Mapping

Expectations. As stated, we expect the JAM to be weaker than ARMv8 since the JAM is a language memory model which is subject to aggressive compiler optimizations. That is, we expect that any time ARMv8 exhibits a behavior the JAM should too and there should be instances where the JAM exhibits behaviors that ARMv8 doesn't. The lone exception is cases where the broad definition of causal cycles adopted by the JAM will rules out behavior like the load-buffering example of Figure 3.2c.

Results. Aside from `totalco` and `LB`, all the tests come from the supplementary material accompanying the ARMv8 model of [74] which we use for comparison. Figure 3.14 shows the results of our comparison and they agree with our expectations: the JAM is at least as weak as ARMv8 except in the case of load-buffering (LB).

Many of the results owe to the fact that the JAM is not multi-copy atomic. That is, unlike ARMv8, different threads can see writes to different locations in different orders. So, `IRIW+*` and `WRC+*` are allowed for the JAM but not for ARMv8. The `WW+RR+WW+RR+wsilp+poaa+wsilp+poaa` litmus test is a variant of IRIW where all writes are release writes and all reads are aquire reads. The writes in this test use a weaker form of synchronization than the example in Figure 3.12a in Section 3.4.4 where the behavior is allowed by the JAM. As a result this behavior is allowed by the JAM. Finally, the LB test is identical to the example in Figure 3.2c from Section 3.3. This execution is allowed by the ARMv8 model but it is a cycle in `(po | rf) & opq` which is explicitly disallowed by the JAM.

JAM Relationship. Taken together this means that the JAM, with its acyclic causality re-

quirement, is incomparable to ARMv8 in terms of allowed behaviors and executions. This appears as the lack of a relationship between the two models in the diagram of Figure 3.1. However, removing the acyclic causality requirement results in the JAM-, which admits strictly more executions than ARMv8. This is represented in Figure 3.1 as the transitive relationship from the JAM- to ARMv8 through C11 and ARMv7. The relationship between ARMv7 and ARMv8 is based on the work of [74]. The relationship between C11 and ARMv7 is based on the conventional wisdom we derive from the construction of C11 compilers. We will establish the relationship between the JAM- and C11 next.

3.5.2 Comparison with RC11

C11's atomic memory accesses can be annotated with memory orders. Herd provides the following built-in relations for the memory orders and accesses in the C11 litmus tests: `M` for plain memory access, `RLX` for relaxed memory order accesses, `REL` for release memory order accesses, `ACQ` for acquire memory order accesses, `REL_ACQ` for release-acquire memory order read-write accesses, `SC` for sequentially consistent memory order accesses, `F & REL` for release fences, `F & AQR` for acquire fences, `F & SC` for sequentially consistent (full) fences, and `F \ SC` for all other fences.

```
let opq = RLX | ACQ | REL | ACQ_REL | SC
let rel = REL | ACQ_REL | SC
let acq = ACQ | ACQ_REL | SC
let vol = SC
let svo = po;[F & REL];po;[W] | [R];po;[F & ACQ];po
let spush = po;[F & SC];po
```

Mapping. Our mapping for the relations provided by Herd for C11 follows the informal relationship outlined in the documentation for the JAM [52]. All plain accesses in C11 are treated as plain accesses in our mapping to the JAM. We map relaxed memory order accesses or stronger to opaque mode, `opq = RLX | ...` , release memory order accesses or stronger to release mode, `rel = REL | ...` , acquire memory order accesses or stronger to acquire mode, `acq = ACQ |`

`...` . sequentially consistent memory order accesses to volatile mode, `vol = SC`. We also map release fences before writes, `po;[F & REL];po;[W]`, and acquire fences after reads, `[R];po;[F & ACQ];po`, to specified visibility orders, `svo`. Finally, we map sequentially consistent fences, `po;[F & SC];po`, to push orders between program order earlier and program order later accesses.

Expectations. The access modes of the JAM are inspired by the memory orders of C11 so this comparison is of particular importance. We expect the JAM to match C11 except in cases where release-sequences, consume-reads, or causal cycles are involved.

Results. In Figure 3.16 we have the results of the comparison with the RC11 model of [47]. We include the tests from research by [88], [85], [47]. In the case of [88] and [85] we used the tests directly. In the case of [47] we translated the tests from the paper to Herd litmus tests ourselves. The results largely agree with our expectations. The exceptions are places where the RC11 model breaks with the C11 specification. We will discuss each in turn.

The `cyc_na` test is the same as the load-buffering example from Section 3.3 but all the accesses are plain. The JAM allows this cycle in plain mode `po | rf` because its acyclicity requirement only applies to opaque mode or stronger accesses. The RC11 model breaks with C11 by including an acyclicity requirement for `po | rf` for all memory accesses.

The `lb` test is the same as `cyc_na` except that all of the accesses are relaxed memory order. In the mapping to the JAM this translates into opaque mode accesses which are not allowed to exhibit a cycle in `po | rf`. Thus both models forbid the load-buffering behavior in this case.

The `mp_relacq_rs` test leverages the release sequences of C11. Since the JAM does not include release sequences it does not forbid the behavior of this test.

Our test runs for `fig6` and `fig6_translated` timed out at 5 minutes. The behavior modeled by these tests highlights a quirk in C11's rules for SC accesses. Together they demonstrate that strengthening the memory order of a particular relaxed store in `fig6` to an SC store in `fig6_translated` creates new behaviors. That is, the tests demonstrate that the memory orders of C11 are not monotonic. RC11 includes a fix proposed by [85] which forbids this behavior. In Section 3.6 we prove that the JAM's access modes are indeed monotonic in our model which means that stronger access modes exhibit fewer behaviors, thus the behavior in these tests is

[85]

name	RC11	JAM
a1	Sometimes	Sometimes
a1_reorder	Sometimes	Sometimes
a3	Sometimes	Sometimes
a3_reorder	Sometimes	Sometimes
a3v2	Sometimes	Sometimes
a4	Never	Never
a4_reorder	Sometimes	Sometimes
arfna	Never	Never
arfna_transformed	Never	Never
b	Never	Never
b_reorder	Sometimes	Sometimes
c	Never	Never
c_p	Never	Never
c_p_reorder	Never	Never
c_pq	Never	Never
c_pq_reorder	Never	Never
c_q	Never	Never
c_q_reorder	Never	Never
c_reorder	Never	Never
cyc	Never	Never
cyc_na	**Never**	**Sometimes**
fig1	Always	Always
fig6	Never	**timed out**
fig6_translated	Never	**timed out**
lb	**Never**	**Never**
linearisation	Never	Never
linearisation2	Never	Never
roachmotel	Never	Never
roachmotel2	Never	Never
rseq_weak	Sometimes	Sometimes
rseq_weak2	Always	Always
seq	Never	Never
seq2	Never	Never
strengthen	Never	Never
strengthen2	Never	Never

New C11 Tests

name	RC11	JAM
IRIW-sc-rlx-acq	**Sometimes**	**Never**

[88]

name	RC11	JAM
cppmem_iriw_relacq	Sometimes	Sometimes
cppmem_sc_atomics	Never	Never
iriw_sc	Never	Never
mp_fences	Never	Never
mp_relacq	Never	Never
mp_relacq_rs	**Never**	**Sometimes**
mp_relaxed	Sometimes	Sometimes
mp_sc	Never	Never

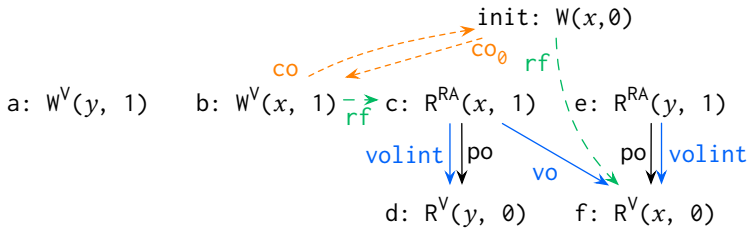
[47]

name	RC11	JAM
2+2W	Never	Never
IRIW-acq-sc	**Sometimes**	**Never**
RWC+syncs	Never	Never
W+RWC	Never	Never
Z6.U	**Sometimes**	**Never**

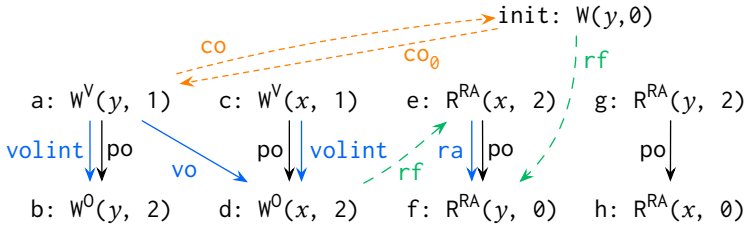
Herd X86 Tests

name	x86	JAM
4.SB	Sometimes	Sometimes
6.SB	Sometimes	**timed out**
6.SB+prefetch	Sometimes	**timed out**
CoRWR	Never	Never
iriw-internal	Sometimes	Sometimes
iriw	Never	Sometimes
podrw000	Sometimes	Sometimes
podrw001	Sometimes	Sometimes
SB	Sometimes	Sometimes
SB+mfences	Never	Never
SB+rfi-pos	Sometimes	Sometimes
SB+SC	Sometimes	Sometimes
X000	Sometimes	Sometimes
X001	Sometimes	Sometimes
X002	Sometimes	Sometimes
X003	Sometimes	Sometimes
X004	Sometimes	Sometimes
X005	Sometimes	Sometimes
X006	Sometimes	Sometimes
x86-2+2W	Never	Sometimes

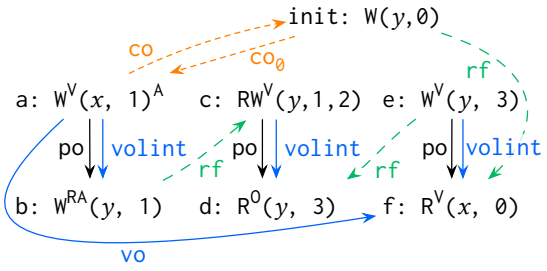
Figure 3.16: RC11 & x86 Litmus Test Comparison



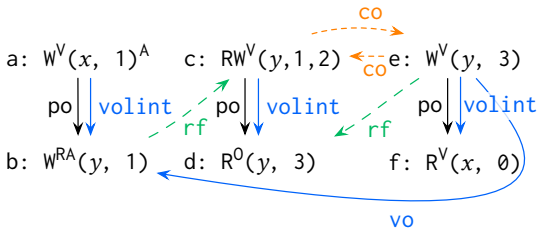
(a) IRIW RA and Volatile Reads



(b) IRIW Extra Writes



(c) Z6.U a $\xrightarrow{vo}$ f



(d) Z6.U e $\xrightarrow{vo}$ b

Figure 3.17: Z6.U & IRIW RA with Volatile Reads Litmus Tests

forbidden.

In the case of `IRIW-acq-sc`, `Z6.U`, and `IRIW-sc-rlx-acq` our model forbids the behavior in keeping with the C11 specification. We will consider each case and discuss why RC11 does not forbid each behavior.

The `IRIW-acq-sc` test appears in in Figure 3.17a. The reason this is forbidden in our model is that all accesses before a SC/Volatile read are ordered by `volint`. Then, because there are two such orders in the reading threads there is either a visibility order, c $\xrightarrow{vo}$ f or d $\xrightarrow{vo}$ e. Thus, one of the two reads must see the non-initialization write using the same reasoning from Section 3.4.4 for Figure 3.12b. This effectively emulates the "leading" fence compilation scheme described in [47], where a full fence is placed before the SC/Volatile accesses. The authors (personal commu-

nication) point out that this scheme should forbid this behavior. By contrast RC11 relaxes the C11 model to accommodate a leading *or* trailing fence compilation scheme. In this case, if the trailing fence scheme is used there's no ordering provided to any of the SC accesses and the execution is allowed.

Two possible executions for `Z6.U` test appears in Figures 3.17c and 3.17d. They represent each case of the visibility orders induced by a $\xrightarrow{\texttt{volint}}$ b and e $\xrightarrow{\texttt{volint}}$ f. In 3.17c we have a $\xrightarrow{\texttt{vo}}$ f, which means that f must have seen a and could not read from the initialization by `cowr`. In 3.17d we have e $\xrightarrow{\texttt{vo}}$ b. We also have b $\xrightarrow{\texttt{rf}}$ c. Then together we have e $\xrightarrow{\texttt{vo}}$ b $\xrightarrow{\texttt{rf}}$ c. Then because c and e are both writes to y we have e $\xrightarrow{\texttt{co}}$ c by `coww`. Separately, since c $\xrightarrow{\texttt{volint}}$ d we have c $\xrightarrow{\texttt{vo}}$ d. Then since e $\xrightarrow{\texttt{rf}}$ d we have c $\xrightarrow{\texttt{co}}$ e by `cowr`. Thus we have a cycle in co and the execution if forbidden.

This effectively emulates the "trailing" fence compilation scheme described by [47], where a full fence is placed after SC/Volatile accesses. They point out that this scheme should forbid this behavior. Again, RC11 relaxes C11 to model both schemes and under a leading fence compilation scheme the behavior this allowed.

Finally, in Figure 3.17b we have our own variant of IRIW called `IRIW-sc-rlx-acq` based on the `WW+RR+...` test in the ARMv8 suite. In this case, if the compiler is inserting trailing fences after a and c, then either a $\xrightarrow{\texttt{vo}}$ d or b $\xrightarrow{\texttt{vo}}$ c. Taking the first case we can can construct b $\xrightarrow{\texttt{vo+}}$ f from a $\xrightarrow{\texttt{vo}}$ d $\xrightarrow{\texttt{rf}}$ a $\xrightarrow{\texttt{ra}}$ f and a cycle in co by `cowr`. The second case is similar but with the read of x in the 4th thread.

Again, the RC11 model does not forbid this execution. In our discussion with the authors, they suggested two possible interpretations for this behavior. The first is a leading fence compilation scheme which we have discussed above. The second is a merge of each pair of writes into a single write. In the second case we suggest that the Java compiler should not merge opaque mode (or stronger) accesses.

In summary, our model correctly forbids the behavior in each of these cases. The reason is that volatile reads would be preceded by a full fence and volatile writes would be followed by a full fence in the presence of mixed mode programs with SC accesses. Importantly, the tension

between optimal compilation and a simple model exists here as it does with C11.

JAM Relationship. Where the diagram of Figure 3.1 is concerned every litmus test, suggests that the JAM is weaker than RC11. Further, since the C11 specification does not rule out causal cycles if we remove the acyclic causality requirement the JAM it is strictly weaker than the C11 specification. Thus JAM- appears below C11 in the diagram.

3.5.3 Comparison with x86

Herd provides the following built in relations for x86 litmus tests: `M` for memory accesses, `SFENCE` for fences for intra-thread ordering of writes with other accesses, `LFENCE` for ordering intra-thread ordering of reads with other accesses, and `MFENCE` as a full fence with cross thread ordering guarantees.

```
let opq = M
let svo = [W];po;[SFENCE];po | [R];po;[LFENCE];po
let spush = po;[MFENCE];po
```

Mapping. We map from all regular memory accesses to opaque mode `opq = M` in keeping with the opaque mode mapping from our comparison with ARMv8. We map `SFENCE` to specified visibility orders from writes to other memory accesses across the fence, `[W];po;[SFENCE];po`. We map `LFENCE` to specified visibility orders from reads to other memory accesses across the fence, `[R];po;[LFENCE];po`. Finally we map `MFENCE` as a specified push orders between any access before the fence to any access after the fence, `po;[MFENCE];po`.

Expectations. We expect the JAM to be weaker than x86 in all cases. The only weak behavior that x86 exhibits is reordering writes with reads (store-buffering) which the JAM also allows in opaque mode.

Results. Figure 3.16 shows the results of our comparison with the x86 model included with Herd. The litmus tests are also included with Herd for the model. The two tests which timed out, `6.SB+prefetch` and `6.SB`, are store buffering variants. Given that the JAM can reorder writes with reads the behavior of these tests is allowed. Otherwise the tests confirm our expectations.

JAM Relationship. Each of the test outcomes demonstrates that there are strictly more executions allowed by the JAM and therefore JAM-. In particular, causal cycles are impossible on x86 because reads cannot be reordered with writes.

3.6 Metatheory

Here, we develop a metatheory for our model of the JAM. First, we detail the semantics and then sketch the proofs for the theorems listed in Section 3.2. We show that each mode, from Volatile to Opaque, admits strictly more executions, otherwise referred to as monotonicity. We also show that properly synchronized programs, i.e. race-free programs as defined by [13], will only exhibit sequentially consistent behaviors as required by Java's data-race-free (DRF) guarantee. Finally, we show that acquire mode for all reads obviates the inclusion of the JAM's acyclic causality requirement.

These theorems further validate the definitions of our model, give more evidence that the model is complete with respect to the documentation, and clearly demonstrate that the semantics is suitable for formal reasoning. The semantics, lemmas and theorems have been mechanized in Coq. The source is available for inspection in the supplementary material and it includes instructions on where to find everything included below.

3.6.1 Semantics

The axioms of our model apply to memory event graphs, each of which represents one possible execution of a program. Thus far we have relied on Herd to generate executions for a program and then apply the axioms of our model to rule on whether an execution is allowed. Our formalization in Coq replaces the Herd machinery with the history fragment of the semantics of [19] to model the set of possible executions.

The syntax of our formalization appears in Figure 4.1. We use n, l, i, and p to range over natural numbers, memory locations, unique memory access identifiers, and unique thread identifiers. We use m to represent one of the four access modes, `P` for plain, `O` for opaque, `RA` for release acquire

$$
\begin{array}{rlrl}
\text{Nat} & n := 0 \mid 1 \mid \ldots & \text{Accesses} & a := [l]_{\mathtt{m}} \mid [l]_{\mathtt{m}} := n \mid \mathsf{RW}(l, n) \\
\text{Locations} & l := \ldots & \text{Mem. Events} & h := \mathsf{init}(i, p) \mid \mathsf{is}(i, a) \mid \mathsf{exec}(i) \\
\text{Modes} & m := \mathsf{P} \mid \mathsf{O} \mid \mathsf{RA} \mid \mathsf{V} & & \mid \mathsf{rf}(i, i) \mid \mathsf{vo}(i, i) \mid \mathsf{push}(i, i) \\
\text{Access Ids} & i := \ldots & \text{Prog. Events} & d := i = a \mid i \mid i \text{ to } n \\
\text{Thread Ids} & p := \ldots & \text{Program} & P := \ldots \\
 & & \text{History} & H := \epsilon \mid H, h
\end{array}
$$

Figure 3.18: Syntax

$$
\frac{P \xrightarrow{d@p} P' \quad H \xrightarrow{d@p} H'}{(P, H) \to (P', H')}\ \text{step}
\qquad
\frac{\mathsf{wf}(H, \mathsf{exec}(i), p) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H, \mathsf{exec}(i)})}{H \xrightarrow{i@p} H, \mathsf{exec}(i)}\ \text{write}
$$

$$
\frac{\neg H(\mathsf{init}(i, \_))}{H \xrightarrow{i=a@p} H, \mathsf{init}(i, p), \mathsf{is}(i, a)}\ \text{init}
\qquad
\frac{\mathsf{wf}(H, \mathsf{rf}(i_w, i), p, n) \quad \begin{array}{r}\mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H, \mathsf{rf}(i_w, i)}) \\ \mathsf{acyclic}(\xrightarrow{\mathsf{po} \mid \mathsf{rf}}_{H, \mathsf{rf}(i_w, i)})\end{array}}{H \xrightarrow{i \text{ to } n@p} H, \mathsf{rf}(i_w, i)}\ \text{read}
$$

$$
\begin{aligned}
\mathsf{reads}(H, i, l) &\triangleq \exists m, H(\mathsf{is}(i, [l]_{\mathtt{m}})) \vee \exists n, H(\mathsf{is}(i, \mathsf{RW}(l, n))) \\
\mathsf{writes}(H, i, l, n) &\triangleq \exists m, H(\mathsf{is}(i, [l]_{\mathtt{m}} := n)) \vee H(\mathsf{is}(i, \mathsf{RW}(l, n))) \\
\mathsf{executed}(H, i) &\triangleq H(\mathsf{exec}(i)) \vee \exists i_w, H(\mathsf{rf}(i_w, i)) \\
\mathsf{executable}(H, i) &\triangleq \neg\, \mathsf{executed}(H, i) \wedge \forall i', i' \xrightarrow{\mathsf{into}}_H i \implies \mathsf{executed}(H, i')
\end{aligned}
$$

$$
\mathsf{wf}(H, \mathsf{exec}(i), p) \triangleq \begin{cases} H(\mathsf{init}(i, p)) \\ H(\mathsf{is}(i, [l]_{\mathtt{m}} := \_)) \\ \mathsf{executable}(H, i) \end{cases}
\qquad
\mathsf{wf}(H, \mathsf{rf}(i_w, i), p, n) \triangleq \begin{cases} H(\mathsf{init}(i, p)) \\ \mathsf{reads}(H, i, l) \\ \mathsf{writes}(H, i_w, l, n) \\ \mathsf{executed}(H, i_w) \\ \mathsf{executable}(H, i) \end{cases}
$$

Figure 3.19: Semantics

and V for volatile. Note that RA writes are release writes and RA reads are acquire reads. Memory accesses, a, can take the form of reads, $[l]_{\mathtt{m}}$, and writes, $[l]_{\mathtt{m}} := n$ with their accompanying modes as well as read-writes, $\mathsf{RW}(l, n)$.

Memory events, h record the program's interactions with memory. Notably, we assume that the program and history can record specified visibility and specified push orders, $\mathsf{vo}(i, i)$ and $\mathsf{push}(i, i)$ before the execution of the related identifiers. For example, the program may use the labeling and ordering mechanism of Crary and Sulivan [19]. We discuss the other events in detail below.

We use d to range over program generated events which are translated by the memory semantics into memory events and we use P to abstract over an expression language that can produce such events. We use H to represent a list of memory events, where $H(h)$ means that $h \in H$. Finally we use $i \xrightarrow{\mathsf{R}}_H i$ to represent memory model relations for H.

Program and history states transition together via the step relation defined in figure 3.19. In Section 4 we will define an expression semantics for programs, P, that generates program events, for now we leave it abstract. Here we focus on the history transition semantics, which certifies program events $d@p$ of the form $i = a@p$, $i@p$, or i to $n@p$ and turns them into memory events.

The program event $i = a@p$ represents the "initialization" of a memory access a using the unique identifier i in thread p. The history semantics appends $\mathsf{init}(i, p)$ and $\mathsf{is}(i, a)$ to H to record the initialization and the form of the memory access identified by i. The only certification required of an initialization is that the memory access identified by i is not already initialized. Importantly, we assume that the program initializes memory accesses in program order, so the subsequence of inititalization events records program order in H.

$i@p$ represents the execution of a write in thread p which is recorded in history with $\mathsf{exec}(i)$. Writes must be certified by the acyclicity requirement for co and a basic well-formedness condition $\mathsf{wf}(H, \mathsf{exec}(i), p)$. The well-formedness requires that i be initialized $H(\mathsf{init}(i, p))$, that the memory access associated with i be a write, $H(\mathsf{is}(i, [l]_{\mathtt{m}} := n))$, and that the write be executable $\mathsf{executable}(H, i)$.

The $\mathsf{executable}(H, i)$ constraint requires that i has not already executed and that the sequence

of execution, to, respects any intra-thread ordering, into, as in the description of Section 3.4.2. The program is otherwise free to execute memory accesses out-of-order. This is in keeping with our restriction on trace order as detailed in Section 3.4.2

i to $n@p$ represents the execution of a read i, reading the value n in thread p which is recorded in history with $\mathsf{rf}(i_w, i)$. Reads must be certified by the same acyclicity requirement for co and the additional requirement on $\mathtt{po} \mid \mathtt{rf}$. The well-formedness condition for reads requires that i be initialized, that it be a read or a read-write, $\mathsf{reads}(H, i, l)$, and that it be executable. It further requires that the paired write i_w is executed, $\mathsf{executed}(H, i_w)$, and that i_w writes the value n to the same location l, $\mathsf{writes}(H, i, l, n)$.

In what follows, the restriction on the trace order, to, and the acyclicity requirements form the interface with the relations of our axiomatic model. Otherwise, the well-formedness conditions from the history rules are unified as an assumption in our theorems called *trace coherence*.

3.6.2 Theorems

To begin, we demonstrate the monotonicity of our access mode definitions. We define the reflexive ordering of the access modes as $\mathtt{P} \sqsubseteq \mathtt{O} \sqsubseteq \mathtt{RA} \sqsubseteq \mathtt{V}$ and extend it to accesses $[l]_{\mathtt{m\_1}} \sqsubseteq [l]_{\mathtt{m\_2}}$, $[l]_{\mathtt{m\_1}} := n_1 \sqsubseteq [l]_{\mathtt{m\_2}} := n_2$, $\mathsf{RW}(l, n_1) \sqsubseteq \mathsf{RW}(l, n_2)$ whenever $m_1 \sqsubseteq m_2$. As a technical matter we treat read-writes as always having the same order. We extend the order to histories by matching identifiers and ordering the accesses.

$$H_1 \sqsubseteq H_2 \triangleq \forall\, i\, a_1\, a_2, H_1(\mathsf{is}(i, a_1)) \wedge H_2(\mathsf{is}(i, a_2)) \Rightarrow a_1 \sqsubseteq a_2$$

When the po, rf, and to relations of two histories H_1 and H_2 have the following relationships: $\xrightarrow{\mathsf{po}}_{H_2} \subseteq \xrightarrow{\mathsf{po}}_{H_1}$, $\xrightarrow{\mathsf{to}}_{H_2} \subseteq \xrightarrow{\mathsf{to}}_{H_1}$, $\xrightarrow{\mathsf{rf}}_{H_2} \subseteq \xrightarrow{\mathsf{rf}}_{H_1}$, then we say they *match*.

Theorem 2 (Monotonicity)**.** *For two histories H_1 and H_2, suppose that both match, both are trace coherent, and $H_2 \sqsubseteq H_1$. Further suppose that* $\mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H_1})$ *and that there are no specified visibility orders or push orders in H_2, then* $\mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H_2})$

We make two notes. First the absence of specified orders in H_2 is a technical convenience since specified order edges are not related to the strength of the access modes for reads and

writes. Second, we focus on the acyclic coherence requirement because the match assumption means that it would be trivial to satisfy the acyclic causality requirement for H_2 supposing it is true of H_1 because po and rf have fewer edges in H_2.

Proof Sketch. We assume $i \xrightarrow{\mathsf{co}}_{H_2} i$ for some i and show that this must mean $i \xrightarrow{\mathsf{co}}_{H_1} i$ which is a contradiction. This is straight forward by induction on $i \xrightarrow{\mathsf{co}}_{H_2} i$, noting that each case of visibility in H_2 will exist in H_1 because of stronger access modes in H_1. □

For the last two theorems will first establish that the main component of visibility in our model, vvo, is irreflexive.

Lemma 1 (Irreflexive Visibility)**.** *If H is trace coherent then, for all i, $\neg\, i \xrightarrow{\mathsf{vvo+}}_H i$.*

Proof Sketch. This follows from two facts. First, vvo derives from orderings that are always either program ordered from intra-thread synchronization (svo, spush, ra, volint) or trace ordered (pushto, rf). Second, both relations are total and to is consistent with the po edges for intra-thread visibility by the executable well-formedness condition. □

Next we show that if a program is properly synchronized then it will only exhibit sequentially consistent behavior. We require the following standard definitions including the traditional notion of sequential consistency [78]:

$$i_1 \xrightarrow{\mathsf{fr}}_H i_2 \triangleq \exists i_3, i_3 \xrightarrow{\mathsf{rf}}_H i_1 \wedge i_3 \xrightarrow{\mathsf{co}}_H i_2$$

$$i_1 \xrightarrow{\mathsf{com}}_H i_2 \triangleq i_1 \xrightarrow{\mathsf{co}}_H i_2 \vee i_1 \xrightarrow{\mathsf{rf}}_H i_2 \vee i_1 \xrightarrow{\mathsf{fr}}_H i_2$$

$$i_1 \xrightarrow{\mathsf{sc}}_H i_2 \triangleq i_1 \xrightarrow{\mathsf{po}}_H i_2 \vee i_1 \xrightarrow{\mathsf{com}}_H i_2$$

We also require a definition of proper synchronization in keeping with our focus on visibility.

$$i_1 \xrightarrow{\mathsf{sync}}_H i_2 \triangleq \exists i_3, i_1 \xrightarrow{\mathsf{vvo+}}_H i_3 \xrightarrow{\mathsf{po*}}_H i_2 \wedge \forall i_4, i_3 \xrightarrow{\mathsf{po*}}_H i_4 \Rightarrow i_3 \xrightarrow{\mathsf{vvo}}_H i_4$$

The idea is that any conflicting access is visibility ordered by some mechanism, be it a specified order (fence) or a strong mode for i_3. Then, following the definition of "type 2" data-races from [13], we say that H is *race-free* when, for all conflicting accesses i_1 and i_2, we have $i_1 \xrightarrow{\mathsf{sync}}_H i_2$ or $i_2 \xrightarrow{\mathsf{sync}}_H i_1$.

Theorem 3 (DRF-SC)**.** *If H is trace coherent, race free and,* acyclic($\xrightarrow{\texttt{co}}_H$)*, then* acyclic($\xrightarrow{\texttt{sc}}_H$)*.*

Proof Sketch. We assume $i \xrightarrow{\texttt{sc+}}_H i$ for some i and show a contradiction. By Lemma 1 it is enough to demonstrate a cycle in vvo+.

We can show that for any i_1 and i_2, if we have $i_1 \xrightarrow{\texttt{com}}_H i_2$ then we have $i_1 \xrightarrow{\texttt{sync}}_H i_2$. Note that any accesses related by com are conflicting. Then we have either $i_1 \xrightarrow{\texttt{sync}}_H i_2$ or $i_2 \xrightarrow{\texttt{sync}}_H i_1$. In each case for com we can show that $i_2 \xrightarrow{\texttt{sync}}_H i_1$ creates a cycle in the coherence order so it must be $i_1 \xrightarrow{\texttt{sync}}_H i_2$.

Then, since po is irreflexive, we have that any sequence $i \xrightarrow{\texttt{sc+}}_H i$ must include at least one com edge. Then since com edges are also sync edges, when we have $i_1 \xrightarrow{\texttt{sc+}}_H i_2$ we also have $i_1 \xrightarrow{\texttt{po}\mid\texttt{com+}}_H i_2$ and therefore $i_1 \xrightarrow{\texttt{po}\mid\texttt{sync+}}_H i_2$.

But then we can rearrange a cycle in $i \xrightarrow{\texttt{po}\mid\texttt{sync+}}_H i$ to be $i \xrightarrow{\texttt{sync+}}_H i$ by appending a leading po edge to the end. Observe that for any sequence $i_1 \xrightarrow{\texttt{sync+}}_H i_2$ we have $i_1 \xrightarrow{\texttt{vvo+}}_H i_2$, so we have $i \xrightarrow{\texttt{vvo+}}_H i$ as required. □

Finally, we show that if all reads are acquire-reads then the JAM's acyclic causality requirement is unnecessary. This theorem demonstrates the soundness of proposed compiler implementations for satisfying the acyclic causality requirement of the JAM [68].

Theorem 4 (Causal Acquire-Reads)**.** *If H is trace coherent and all reads in H are acquire-reads, then* acyclic($\xrightarrow{\texttt{po}\mid\texttt{rf}}_H$)*.*

Proof Sketch. We assume $i \xrightarrow{\texttt{po}\mid\texttt{rf+}}_H i$ for some i and show a contradiction. By Lemma 1, it is enough to demonstrate a cycle in vvo+.

First, note that any sequence $i_1 \xrightarrow{\texttt{rf}}_H i_2 \xrightarrow{\texttt{po}}_H i_3$ implies $i_1 \xrightarrow{\texttt{vvo+}}_H i_3$ because rf implies vvo and because, by assumption, the read i_2 is an acquire read and i_3 is program order later, so $i_2 \xrightarrow{\texttt{vvo}}_H i_3$.

Let $\xrightarrow{\texttt{rfpoq}}_H$ be a reads-from edge followed by an optional program order edge. Note that, by induction and the fact that rf and rfpo imply vvo we can show that $i_1 \xrightarrow{\texttt{rfpoq}}_H i_2$ implies $i_1 \xrightarrow{\texttt{vvo+}}_H i_2$

Then, since po is irreflexive, we have that any sequence $i \xrightarrow{\texttt{po|rf+}}_H i$ must include at least one rf edge. Thus we can rearrange to get $i' \xrightarrow{\texttt{rfpoq+}}_H i'$ and we have $i' \xrightarrow{\texttt{vvo+}}_H i'$ by the above, as required. □

3.7 Unobservable Total Coherence Order

Here we will demonstrate that the effects of a total coherence order are unobservable in our model. This is a surprising result since a total coherence order is intuitively associated with maintaining per-location state and we would expect concurrent writes to have a material impact on the behavior of programs.

To start, recall that a feature of a memory model as exhibited by an example program is defined in binary terms. The feature is present when any executions are allowed and the feature is absent when no executions are allowed. Then we say that a feature is *observable* when we can construct any example program such that the feature's presence in the model changes whether any executions are allowed. So, to observe a total coherence order, we must construct a program that changes whether executions are allowed based on the presence of the total order. We will show that this is impossible under our model.

First, note that in our model there is no way to allow previously forbidden executions when adding coherence order edges. This means we can't observe the feature by going from some executions *with* a total order to no executions *without* a total order. Thus, any program that allows us to observe the total coherence order must forbid all executions in the presence of a total order. We will show that such a program will also forbid all executions when the total order is removed.

Theorem 5. *It is impossible to construct an example program for the JAM, such that, for any two writes, if they are totally ordered there are no valid executions, and if they are not totally ordered there is at least one valid execution.*

Proof. From a total order we have that all candidate executions can be divided between the two

directions. For some writes to the same location a and b we have:

$$a \xrightarrow{co} b \vee \tag{3.1}$$

$$b \xrightarrow{co} a \tag{3.2}$$

We begin by eliminating all executions for the first direction, (1). This means we must construct a cycle in co, such that, for all executions including (1) we have :

$$a \xrightarrow{co} b \tag{1}$$

$$b \xrightarrow{co} \cdots \xrightarrow{co} a \tag{3.3}$$

With this cycle, executions including (1) are forbidden by the acyclicity requirement for co. Observe that (3) must be present in all executions of the program. If it is not then we can divide the executions that do not have (3) between executions with (1) and executions with (2). For the executions without (3) and with (1) the execution is allowed and we have failed to forbid all executions. Now we must eliminate all executions for (2) with another cycle:

$$b \xrightarrow{co} a \tag{2}$$

$$a \xrightarrow{co} \cdots \xrightarrow{co} b \tag{3.4}$$

Moreover (4) must persist for all executions for the same reason that (3) persisted. Then it must be that, for all executions we have:

$$b \xrightarrow{co} \cdots \xrightarrow{co} a \tag{3}$$

$$a \xrightarrow{co} \cdots \xrightarrow{co} b \tag{4}$$

Then, even if we remove the total order, (1) and (2), from the model, all executions of the program will still have a forbidden cycle by (3) and (4) and we can not demonstrate a single execution that is allowed. □

Critically, this line of reasoning depends on the fact that we have a single acyclicity requirement in which co participates and both (3) and (4) form a cycle that violates that requirement.

If that were not the case then (3) and (4) would not necessarily create a cycle or forbid any executions. As a result this reasoning does not apply to the ARMv8 and RC11 models as they have multiple acyclicity requirements involving the coherence order. In Appendix E we have included a litmus test for ARMv8 that shows it is possible to construct a program for that model that behaves differently with and without the total order.

Demonstrating that a total coherence order is not observable in our model means that the JAM may adopt a total coherence order without affecting the outcomes of the model. This would allow it to emulate other mainstream memory models in this respect and recover the intuitive notion of per-location state.

3.8 Summary

Having constructed and validated a suitable model from which to perform reasoning with specified orders we can now prove the correctness of concurrent algorithms with the understanding that our proofs will be sound for any stronger memory model.

CHAPTER 4

Proving Correctness with Specified Orders

Here we will present a logic for proving algorithm correctness in the context of Java's opaque mode (hereafter "the JOM") without the guarantee of acyclic causality. As we have seen in Section 3.4, opaque mode is the weakest of the JAM's access modes intended for lock-free algorithms. Beyond acyclic causality, the JOM provides only the most basic guarantees in the form of coherence. Since we are explicitly discarding the guarantee of acyclic causality and relying entirely on coherence any theorems about programs we prove will apply to the same programs running on stronger models.

However, more executions should intuitively mean a more difficult task in reasoning. This is where specified orders shine. Because specified orders recover fragments of sequential consistency, we can reason as though the memory model is sequential consistency and then recover the necessary guarantees from SC for the JOM using specified orders.

Additionally, reasoning for the JOM must proceed without any global notion of state since it does not support a total order on non-atomic writes. To accomplish this we use only expression values and the relations of the memory model. In this "stateless" approach, a proof shows that certain pairings of reads and writes are impossible. The proof proceeds by supposing that a pairing is possible and then deriving a contradiction by building a cycle in the coherence order which is forbidden by the semantics of the JOM. We call this process *write elimination*.

For more general and complex proofs, such as our proof of correctness for a two thread ring-buffer, we must structure many such write elimination steps. Our logic accomplishes this through induction over the partial coherence order.

In Section 4.1 we will build on the history semantics of Section 3.6 with a semantics for expressions to generate the program events which are certified by the axioms of our memory model.

In Section 4.2 we will detail a logic that lifts relational reasoning from the execution graphs to programs and enables equational reasoning for expression values. In Section 4.3 we will prove correctness for Dekker's mutual exclusion algorithm as an introduction to the logic and core the concept of write elimination using specified orders. In Section 4.4 we will prove correctness for a two-thread Ringbuffer algorithm and demonstrate how induction over the partial coherence order of the JAM- structures complex proofs.

4.1 Operational Expression Semantics

We model the expressions of the JOM using an imperative operational semantics, where the program generates program events representing the execution of memory accesses and those events are validated by the axioms of the memory model before being recorded in history as memory events.

We use a core language, detailed in Figure 4.1. Figure 4.2 focuses on the most relevant portion of the semantics. The full expression semantics is available in Appendix F. As in Section 3.4, the top level state step coordinates steps of the program with steps in a sequence of events, a history, which represents memory (rule: step).

The expression semantics for operators, conditionals, and `repeat` loops is standard. We focus here on the rules that most directly affect the memory model. There are three memory access expressions: reads $[s]$, writes $[s] := s$, and read-writes $\mathsf{RW}(s, s)$. Note that we omit the access mode annotation under that assumption that all reads and writes are opaque mode. With that, a memory access goes through three phases: variable substitution, initialization, and execution.

Execution may proceed out of order (rule: let-right) but initialization proceeds in program order by requiring that all memory accesses in the first of the two sequenced expressions is initialized, e_1 init, and that accesses cannot be initialized before all variable arguments are substituted, $x \notin FV(d)$). The initialization phase reduces any access to a unique identifier and records the form of the access in an event $i = a$ (rule: access-init). The uniqueness of the identifier which replaces the access is guaranteed by the history semantics.

Nat	$n := 0 \mid 1 \mid \ldots$	Exp	$e := s \mid s + s \mid s \bmod n \mid s == s$
Value	$v := n$		$\mid \texttt{let}\ x := e\ \texttt{in}\ e \mid \texttt{repeat}\ e\ \texttt{end}$
Labels	$l := \ldots$		$\mid \texttt{if}\ s\ \texttt{then}\ e\ \texttt{else}\ e \mid s = n\ \texttt{in}\ e \mid l{:}e$
Label List	$L := \epsilon \mid l, L$	Prog. Events	$d := i = a \mid i \mid i\ \text{to}\ n$
Simple	$s := v \mid x$	Thread Ids	$p := \ldots$
Accesses	$a := [l] \mid [l] := n \mid \text{RW}(l, n)$	Program	$P := \epsilon \mid P; p : \texttt{fork}\ e$

Figure 4.1: Imperative Syntax

$$\frac{P \xrightarrow{d@p} P' \quad H \xrightarrow{d@p} H'}{(P, H) \to (P', H')}\ \text{step} \qquad \frac{e_2 \xrightarrow{d} e_2' \quad e_1\ \text{init} \quad x \notin FV(d)}{\texttt{let}\ x := e_1\ \texttt{in}\ e_2 \xrightarrow{d} \texttt{let}\ x := e_1\ \texttt{in}\ e_2'}\ \text{let-right}$$

$$\frac{}{a \xrightarrow{i=a} i}\ \text{access-init} \qquad \frac{}{i \xrightarrow{i\,\text{to}\,n} n}\ \text{exec-read} \qquad \frac{}{i \xrightarrow{i} 0}\ \text{exec-write}$$

$$\frac{}{[s/x]e \xrightarrow{\emptyset} s = n\ \texttt{in}\ [n/x]e}\ \text{spec} \qquad \frac{e \xrightarrow{d} e'}{s = n\ \texttt{in}\ e \xrightarrow{d} s = n\ \texttt{in}\ e'}\ \text{spec-under} \qquad \frac{}{n = n\ \texttt{in}\ e \xrightarrow{\emptyset} e}\ \text{spec-rm}$$

Figure 4.2: Semantics

	expression	history
	$\texttt{let } v := [l] \texttt{ in } [l] := v$	H
$\xrightarrow{i_r = [l]}$	$\texttt{let } v := i_r \texttt{ in } [l] := v$	$H,\ \mathsf{init}(i_r, [l])$
$\xrightarrow{i_r \text{ to } 1}$	$\texttt{let } v := 1 \texttt{ in } [l] := v$	$H,\ \mathsf{init}(i_r, [l]), \mathsf{rf}(i_0, i_r)$
$\xrightarrow{\varnothing}$	$[l] := 1$	$H,\ \mathsf{init}(i_r, [l]), \mathsf{rf}(i_0, i_r)$
$\xrightarrow{i_w = ([l] := 1)}$	i_w	$H,\ \mathsf{init}(i_r, [l]), \mathsf{rf}(i_0, i_r),\ \mathsf{init}(i_w, [l] := 1)$
$\xrightarrow{i_w}$	0	$H,\ \mathsf{init}(i_r, [l]), \mathsf{rf}(i_0, i_r),\ \mathsf{init}(i_w, [l] := 1), \mathsf{exec}(i_w)$

(a) Initialization

	...	
$\xrightarrow{i_r = [l]}$	$\texttt{let } v := i_r \texttt{ in } [l] := v$	$H,\ \mathsf{init}(i_r, [l])$
$\xrightarrow{\varnothing}$	$\texttt{let } v := i_r \texttt{ in } v = 1 \texttt{ in } [l] := 1$	$H,\ \mathsf{init}(i_r, [l])$
$\xrightarrow{i_w = [l] := 1}$	$\texttt{let } v := i_r \texttt{ in } v = 1 \texttt{ in } i_w$	$H,\ \mathsf{init}(i_r, [l]),\ \mathsf{init}(i_w, [l] := 1)$
	...	

(b) Speculation

Figure 4.3: Example Execution

Memory accesses are executed, possibly out of program order, by choosing an arbitrary natural number value (rules: exec-read or exec-write). The program event, d, generated by memory access execution is passed down to the history semantics to ensure that the access associated with the identifier can actually produce the chosen value and event. Note that memory locations are addressed directly by natural numbers and we do not consider allocation.

Finally, the expression semantics incorporates a flexible form of speculation. Speculation works by guessing values for free variables in expressions and adding a constraint that requires the substitution of that variable to confirm the guessed value (rules: spec and spec-rm). Since execution can proceed under the constraint (rule: spec-under) the semantics is able to emulate weak behavior introduced by architecture and compiler optimizations without redefining control structure semantics. For example, most modern processors speculatively execute conditionals and later revert the subsequent execution if the wrong branch was chosen.

Figure 4.3a shows an example execution for two sequenced accesses to the same location. First, the read of the concrete location l is initialized to i_r (note that we omit $\mathrm{is}(i, a)$ events for simplicity). Then the read executes and reads from some write i_0 producing the value 1. Once the variable v is substituted into the write it can also initialize and execute.

Alternately, after the first step but before the read executes, the semantics could speculate on the value of v, as in Figure 4.3b. This allows the write to initialize, and possibly even execute, under the assumption that the read can later satisfy the constraint on the value of v. However the read cannot be paired with the write in this example as it would violate the corw coherence rule of the JOM.

4.2 Logic

Our logic abstracts over the executions of the JOM in two ways. For expressions, assertions take the form of innequalities over the final value of expressions. Deduction relies on the transitive properties of inequality of natural number values and the equalities inherent in basic control structures. For example, a `let` expression will eventually reduce to the same value as its second child expression. For memory accesses, assertions take the form of the relations of the memory

model and deduction proceeds along the lines of previous reasoning about example programs.

In the case of expressions this abstraction represents a significant reduction in proof effort. The expression semantics of our model tracks program order, executes memory accesses out-of-order, and emulates compiler optimizations with speculation, all while maintaining a loose coupling to the history semantics via memory access identifiers. As a consequence, reasoning about expressions directly with the semantics would be tedious, requiring an accounting of the many possible outcomes of the step relation for each expression. Moreover, associating memory access expressions to their identifiers and behaviors in the memory model would be difficult.

Here, we detail the assertion language and the semantics of assertions in our logic. In Sections 4.3 and 4.4 we will state and prove theorems using our assertions and the rules of our logic.

4.2.1 Assertions and Deductive Rules

Our assertion language, depicted in Figure 4.4a, is comprised of a standard first order fragment with quantification for labels and natural numbers, $\forall V, A$, and three core program assertions: expression locations, $e \mathbin{@} L$, expression relations, $V \mathbin{\dot{\circ}} V$, and memory relations $L \xrightarrow{\mathsf{R}} L$. We will consider each in turn and give their logical semantics in the next section.

To reason about a program we need a way to identify specific expressions within the program. Further we want to refer to all "instances" of a particular expression, such as library procedures, so that we can reason about all occurrences of the expression. To that end the logic assumes a labeling discipline where all expressions are located with unique label sequences, L. Consider the example program in Figure 4.4d. There, we can locate the read of y with $[y] \mathbin{@} l_1$. Then, if we wish to prove properties about the read that are general with respect to some larger program in which it appears we can quanitfy over the label sequence that locates the parent `let` expression. For example, where the parent `let` expression is located at L we might have $\forall L, [y] \mathbin{@} L, l_1 \Rightarrow P$, where P is some assertion in our assertion language.

Once we have located an expression within a program, we reason about its behavior through the values it produces when it has fully executed. In particular, we can use the same label sequence from the initial state of the program to track the expression's execution until it results in a natural

$$
\begin{array}{rrl}
\text{Expressions} & e := & \ldots \\
\text{Naturals} & n := & \ldots \\
\text{Labels} & l := & \ldots \\
\text{Executions} & E := & E;(P,H) \mid (P,H) \\
\text{Label Seq.} & L := & L;l \mid l \\
\text{Val. Variables} & V := & L \mid n \\
\text{Operators} & \circ := & = \mid < \\
\text{Relations} & R := & \mathsf{co} \mid \mathsf{coi} \mid \mathsf{vo} \mid \mathsf{po} \mid \\
 & & \mathsf{push} \mid \mathsf{rf} \mid \mathsf{vvo} \\
\text{Assertions} & A := & \mathsf{false} \mid A \Rightarrow A \mid \forall V,\ A \mid \\
 & & e\ @\ L \mid V \mathrel{\dot\circ} V \mid L \xrightarrow{\mathsf{R}} L \\
\text{Assumptions} & \Gamma := & \Gamma; A \mid A
\end{array}
$$

(a) Language

$$
\begin{array}{rl}
E \vDash e\ @\ L \Leftrightarrow & \mathsf{match}(E, L, e) \\
E \vDash L \doteq n \Leftrightarrow & E = E'; E'' \wedge E'' \vDash n\ @\ L \\
E \vDash n \mathrel{\dot\circ} n \Leftrightarrow & n \circ n \\
E \vDash L \mathrel{\dot\circ} n \Leftrightarrow & \exists n', L \doteq n' \ \wedge\ n \circ n' \\
E \vDash L_1 \mathrel{\dot\circ} L_2 \Leftrightarrow & \exists n_1, E \vDash L_1 \mathrel{\dot\circ} n_1 \wedge \\
 & \exists n_2, E \vDash L_2 \mathrel{\dot\circ} n_2 \wedge \\
 & n_1 \circ n_2 \\
E \vDash L_1 \xrightarrow{\mathsf{R}} L_2 \Leftrightarrow & \exists i_1, \mathsf{accessid}(E, L_1, i_1) \wedge \\
 & \exists i_2, \mathsf{accessid}(E, L_2, i_2) \wedge \\
 & i_1 \xrightarrow{\mathsf{R}}_E i_2 \\
E \vDash A_1 \Rightarrow A_2 \Leftrightarrow & \text{if } E \vDash A_1 \text{ then } E \vDash A_2 \\
E \vDash \forall V, A \Leftrightarrow & \forall V',\ E \vDash [V'/V]A
\end{array}
$$

(b) Semantics

$$\Gamma \vDash A \ \Leftrightarrow\ \forall E, (\forall A' \in \Gamma, E \vDash A') \Rightarrow E \vDash A$$

(c) Models with Assumptions

$$\texttt{let}\ x := (l_1\texttt{:[}y\texttt{]})\ \texttt{in}\ (l_2\texttt{:}x)$$

(d) Example Program

Figure 4.4: Logical Assertion Language and Semantics

number. In the example program of Figure 4.4d, we can initially locate the local variable x using the label sequence L, l_2. Now, assume that the read returns the value 1. Then we know that x will be replaced with the value 1 by substitution in a later state of the program and we can locate 1 with the same label sequence. We record this fact as the expression equality $L, l_2 \doteq 1$.

Most often we will use the natural equalities inherent between parent and child expressions in our deductions. For example, any `let` expression will eventually reduce to the final value of its second expression. So, from the example program where the parent `let` expression is located at L, we know that $L \doteq L, l_2$. This appears as the rule let-right in Figure 4.5. Then, if we have the equality, $L, l_2 \doteq 1$ we know that $L \doteq L, l_2 \doteq 1$ and by transitivity of $\doteq$ (which is a theorem) we

$$\frac{(\texttt{if}\ s\ \texttt{then}\ l_1{:}e_1\ \texttt{else}\ l_2{:}e_2)\ @\ L}{(L, l_{cnd} \doteq 1 \wedge L \doteq L, l_1) \vee (L, l_{cnd} \doteq 0 \wedge L \doteq L, l_2)}\ \text{if-alt}$$

$$\frac{(\texttt{let}\ x := l_1{:}e_1\ \texttt{in}\ l_2{:}e_2)\ @\ L}{L \doteq L, l_2}\ \text{let-right}$$

$$\frac{\texttt{let}\ x := l_1{:}e_1\ \texttt{in}\ l_2{:}e_2\ @\ L}{\exists n,\ L, l_1 \doteq n}\ \text{let-left}$$

$$\frac{x\ @\ L, l_2, L' \quad x \in \mathsf{FV}(e_2) \quad \texttt{let}\ x := l_1{:}e_1\ \texttt{in}\ l_2{:}e_2\ @\ L}{L, l_2, L' \doteq L, l_1}\ \text{let-bind}$$

(a) Expression Rules

$$\frac{[s]\ @\ L_r \quad L_r \doteq n \quad L_r, l_{loc} \doteq l}{\exists L_w,\ L_w \xrightarrow{\texttt{rf}} L_r}\ \text{reads-rf}$$

$$\frac{L_1 \xrightarrow{\texttt{push}} L_2 \quad L_3 \xrightarrow{\texttt{push}} L_4}{L_1 \xrightarrow{\texttt{vo}} L_4 \ \vee\ L_3 \xrightarrow{\texttt{vo}} L_2}\ \text{vo-pushes}$$

$$[s] := s\ @\ L_1$$

$$\frac{L_1 \xrightarrow{\texttt{vo}} L_r \quad L_2 \xrightarrow{\texttt{rf}} L_r \quad L_1 \neq L_2}{L_1 \xrightarrow{\texttt{co}} L_2}\ \text{cowr}$$

$$\frac{L_1 \xrightarrow{\texttt{co}} L_2 \quad L_2 \xrightarrow{\texttt{co}} L_1}{\text{false}}\ \text{co-cycl}$$

(b) Memory Rules

Figure 4.5: Example Rules

$$\frac{L_1 \xrightarrow{\texttt{co}} L_2 \qquad L_1 \xrightarrow{\texttt{coi}} L_2 \Rightarrow P\ L_1\ L_2 \qquad \forall L_3, L_1 \xrightarrow{\texttt{co}} L_3 \Rightarrow P\ L_1\ L_3 \Rightarrow L_3 \xrightarrow{\texttt{coi}} L_2 \Rightarrow P\ L_1\ L_2}{P\ L_1\ L_2}\ \text{co-ind}$$

Figure 4.6: Coherence Order Induction

know that $L \doteq 1$.

The memory relations of the semantics are lifted from the identifiers of the history semantics to expressions via label sequences that point to memory accesses. The goal is frequently to establish the relationship between a read and a write expression which enables reasoning about the values of read expressions. For the example program, if we know that the read of y at L, l_1 is paired with some write located at L_w, i.e. $L_w \xrightarrow{\mathtt{rf}} L, l_1$, then we can deduce that $L_w \doteq L, l_1$.

We can also perform reasoning on the relations themselves using the axioms of the memory model. Recall the cowr rule of the semantics in Section 3.4. In Figure 4.5 we have lifted the identifiers of cowr to labels matching the appropriate expressions. We use these rules to establish cycles in the coherence order, and thereby prove a contradiction, for read-write pairings that we wish to eliminate. By eliminating writes from which reads can draw their values, we constrain the values that the read expressions can return and thereby constrain the values that the read expression can be equal to at the expression level. We call this process write elimination and we will discuss it in more detail in Section 4.3.

During the process of write elimination we will nearly always leverage specified orders in some form or another. For example the rule vo-pushes in Figure 4.5 encodes the disjunction in visibility between the heads and tails of two visibility orders as detailed in Section 3.4.2.

Finally, in Figure 4.6, the rule co-ind encodes structural induction with predicates in our assertion language over writes at the label sequences L_1 and L_2. Note that co the relation is transitive. So, to perform induction, we introduce the coi relation to signify that there are no intervening writes (i.e. $\mathsf{coi}^+ = \mathsf{co}$). This rule allows proofs to incorporate existing invariant based reasoning about single memory locations in spite of the partial ordering of writes in the JOM.

4.2.2 Semantics

The semantics of our assertion language, $E \vDash A$, is defined inductively, see Figure 4.4b, where an execution, E, is a finite sequence of program history pairs related by the semantics of the top level transition relation, $(P, H) \rightarrow^* (P', H')$. The match function recursively locates expressions within the initial program, P, of an execution, E, using the label sequence L (See Appendix H for the

definition) . We use this to follow the evolution of the expression, located at L, across an execution to a value with $E \vDash L \circ n$. With that we can reason about binary relations over labels and natural numbers, $E \vDash L \circ n$ and $E \vDash L_1 \circ L_2$. In our proofs of Dekker and RingBuffer we use = and <. The memory relations, $L_1 \xrightarrow{\mathsf{R}} L_2$, are also defined in terms of matching expressions. Recall that each memory operation is assigned an identifier i and the relations are defined over these identifiers. The actionid predicate ensures that the labeled expressions reach the correct identifier and that they reach the correct value (See Appendix H for the definition). Other standard connectives are defined in terms of $\Rightarrow$, false, and $\forall V, A$. Finally, we lift our assertions from executions to collections of assertions as in Figure 4.4c.

The careful reader will have noticed that the equational rule for if-alt has a single matching hypothesis of the form e @ L. Further, the semantics of e @ L only match in the original program. In practice all of the expression level rules require an additional assumption of equality, i.e. $\exists n, L \doteq n$ that records the execution of the expression. Since that assumption is identical for every rule we have simply added it to the matching requirement and retained the simpler assertion semantics here for clarity.

4.2.3 Soundness

Where $\Gamma \vdash A$ is defined by the rules of our logic, we can now state our soundness theorem.

Theorem 6 (Soundness)**.** *if* $\Gamma \vdash A$ *then* $\Gamma \vDash A$

The proof proceeds by induction on $\Gamma \vdash A$. Thus, we establish $\Gamma \vDash A$ from the semantic definitions of the assumptions in each rule of the logic.

A full listing of the rules of our logic appears in Appendix G. Proof sketches for the soundness of a few expression and memory ordering rules are included in in Appendix I. Formal proofs of the expression and memory rules appear in the supplementary material.

4.3 Proof by Write Elimination: Dekker

Recall that the JOM only guarantees a per-location partial order on writes. Inuitively, a total order on writes gurantees that each location has a single possible state at any point in time. By contrast

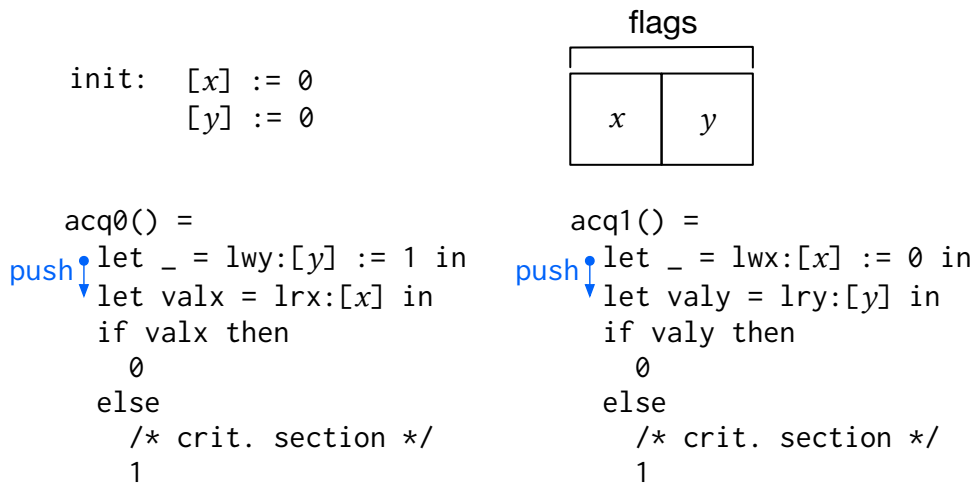


Figure 4.7: Dekker's Mutex

a partial order means there can be many possible states at a given point in time. In the absence of a tradition notion of state, our logic facilitates reasoning about the JOM through a process of *write elimination*. Write elimination removes unwanted writes that a read can be paired with in $\xrightarrow{\texttt{rf}}$. By eliminating such writes we can prove important properties of the value returned by the read without a notion of state.

Our approach to write elimination is based on proof by contradiction. Proofs in this style have three main ingredients: assumptions, derived relations, and derived contradictions. We begin the process of eliminating a write by collecting some basic assumptions including the $\xrightarrow{\texttt{rf}}$ relationship we wish to eliminate. We then add *derived relationships* implied by the memory model and the assumptions. With these relationships we show a contradiction in the form of a cycle in the coherence order.

4.3.1 Dekker

As an example of write elimination we consider Dekker's mutual exclusion algorithm in Figure 4.7. The procedures of the algorithm, executing concurrently, work by communicating their intent to enter the critical section through the flags x and y. We wish to show that it is impossible for both procedures to enter the critical section, see Theorem 7.

To begin, we assume the first thread's read, $[x]$, reads from the initialization and thereby

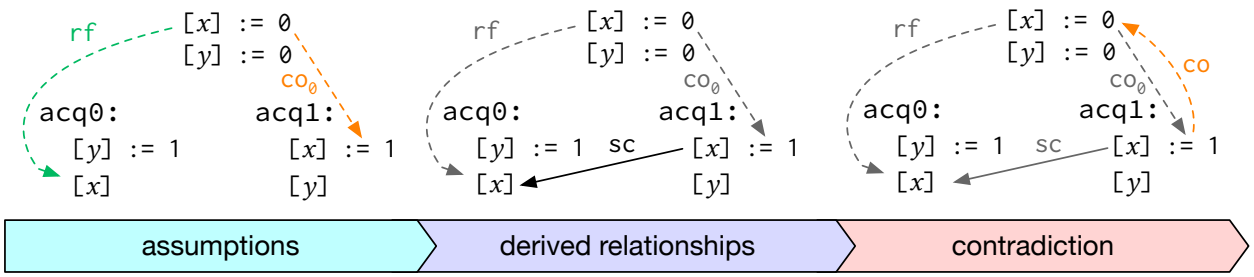


Figure 4.8: First Write Elimination, Dekker, SC

enters the critical section. We wish to eliminate the case where the second thread's read, [y], also reads from the initialization. That is, we want to eliminate the case where both thread's reads see the value 0 for x and y and thereby eliminate the possibility that both threads enter the critical section. If we can show that [y] has "seen" the write, [y] := 1 in acq0 then it could not have read from the initialization because read should always be connected with the latest write it is aware of.

We will first focus on write elimination by examining the proof under sequential consistency and then carry that intuition forward to work with the proof under the JOM.

4.3.2 Write Elimination with Sequential Consistency and the JOM

Recall that sequential consistency provides many relationships between memory accesses. Standard proofs of Dekker (or Dekker like mutex algorithms [35]) rely on small subset of these relationships. In particular these proofs use the execution order inferred from the program order between the read and write of each thread and the total order between [x] in acq0 and [x] := 1 in acq1.

With this small set of relationships in hand we can complete the proof with two write eliminations. We will focus on the first to detail the three step approach to write elimination as set out in Figure 4.8. Note that in this context our reasoning takes place with the memory model relations as in Section 3 but the nodes in our graphs are now the memory access expressions. This is in keeping with the abstraction level of our logic where we reason about program expressions.

First, as stated, we assume that [x] reads from the initialization. We also assume that initialization always happens before any other memory access to the same location. These are the

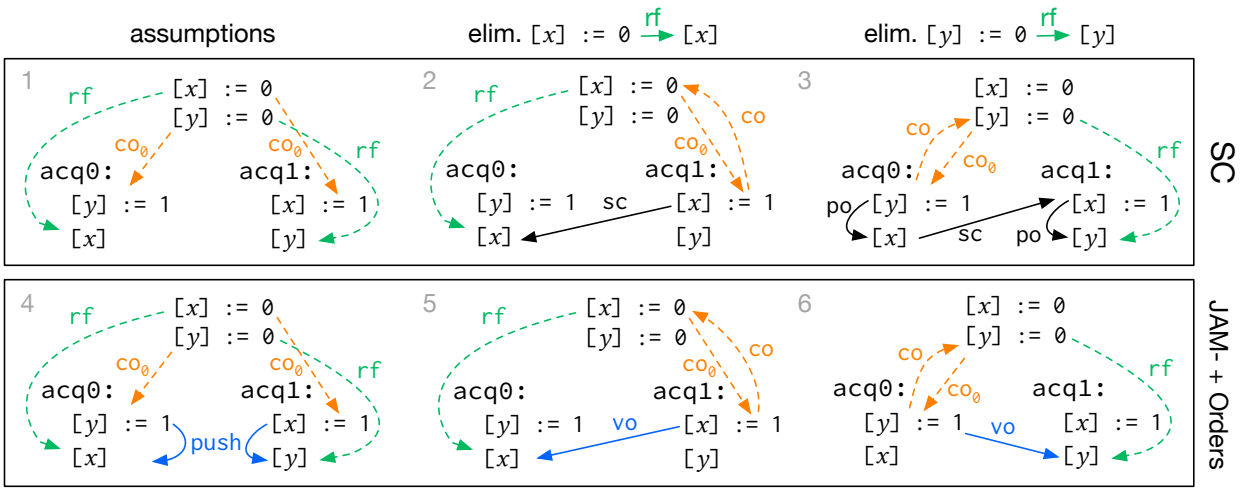


Figure 4.9: All Write Eliminations, Dekker, SC and RMC

edges in Figure 4.8, labeled *assumptions*.

Second, from sequential consistency, we have a total order across threads between `[x]` in `acq0` and `[x] := 1` in `acq1`. One side of the disjunction is shown as the `sc` edge in Figure 4.8. This step is labeled very generally as *derived relations*.

Finally, we construct a contradiction in the form of a cycle in coherence. Recall that in SC, a read should be paired with the latest write it knows about . Thus, if `[x] := 1` executes before `[x]` then `[x]` is aware of `[x] := 1`. Then since the read of x is paired with the initial write it must be the latest of the two writes. We record this as the second contradictory coherence order edge. This step is labeled with *contradiction*. This contradictory cycle eliminates the initialization write from being paired with `[x]` which contradicts this side of the total order between `[x]` and `[x] := 1`.

The same process applies to the second side of the total order derived from sequential consistency between `[x]` in `acq0` and `[x] := 1` in `acq1` as in diagram 3 of Figure 4.9. We assume that `[y]` reads from the initialization. From the right side of the total order we have that `[x]` happens before `[x] := 1`. From program order we have that `[y] := 1` executes before `[x]` and `[x] := 1` executes before `[y]`. Then by transitivity we know that `[y]` must have seen `[y] := 1` and we get the contradictory cycle as illustrated. As a result we know that `[y]` could not have read from the initialization.

Thus, from these two eliminations we know that for both sides of the total order provided by SC one of the reads of x or y can't read from the initialization, as required. Further, since the only other write is the flag write in the opposite thread, one of the reads must see the value 1. Then, for example in the case of the first write elimination, we have that `[`y`]` read the value 1 and we can conclude that the `if` statement must follow the `then` branch. As a result, the procedure would evaluate to 0 and avoid the critical section. Thus, combining the relations provided by sequential consistency allows us to eliminate the unwanted write from the initialization and demonstrate mutual exclusion.

The JOM is far weaker than sequential consistency and it does not provide a way to derive the relationships we needed in the previous proof. For the purposes of Dekker, the only thing that the JOM retains is the axiom that a read should be paired with the latest write it knows about (cowr).

Thus, to perform the two write eliminations we must recover the relationships derived from SC in the write elimination in diagram's 2 and 3 of Figure 4.9. In practice, we do this by specifying a push order between the read and write in both `acq0` and `acq1`.

This manifests in the algorithm as the orders of Figure 4.7. which appear as the assumptions of diagram 4 of Figure 4.9. Recall that a pair of push orders like this provides a cross thread guarantee that either the head of the first push order happens before the tail of the second push order or vice versa (see vo-pushes in Figue 4.5). The result is the derived inter-thread orders in Figure 4.9, diagrams 5 and 6.

We can now consider both cases of the disjunction implied by the push orders. In each case we will derive the same contradiction from the proof under sequential consistency. We assume the `[`x`]` read from the initialization and the additional push orders. From the push orders we derive that `[`x`] := 1` happens before `[`x`]`. Then `[`x`]` must have seen `[`x`] := 1` and, as with sequential consistency, it must be before the initialization which is a contradiction, see diagram 5 in Figure 4.9. Similarly if `[`y`] := 1` is visible to `[`y`]` then `[`y`]` must have seen `[`y`] := 1`. Assuming that it read from the initialization, we derive a cycle and a contradiction, see diagram 6 in Figure 4.9.

$1 \doteq$ `acq0`	by assumption and symmetry	
$\doteq$ `acq0`, l_1	by let-right, `let y := 1 in` $l_1 : \ldots$ @ `acq0`	
$\doteq$ `acq0`, l_1, l_2	by let-right, `let valx := lrx:[`x`] in` $l_2 : \ldots$ @ `acq0`, l_1	(4.1)
$1 \doteq$ `acq0`, l_1, l_2	by (1)	
$\doteq$ `acq0`, l_1, l_2, l_3	by the right side of the left disjunct of if-alt	
$\doteq 0$	contradiction	(4.2)
$0 \doteq$ `acq0`, l_1, l_2, l_{cnd}	by the left side of the right disjunct of if-alt	
$\doteq$ `acq0`, l_1, `lrx`	by let-bind, `let valx := lrx:[`x`] in` $l_2 : \ldots$ @ `acq0`, l_1	(4.3)

Figure 4.10: Equational Reasoning

4.3.3 Expression Equalities

With the basic outline of the proof in place we state our correctness specification for mutual exclusion using expression equalities. Intuitively, since the value of the expression signals whether the method entered the critical section, we can show that if one enters the critical section the other should not.

Theorem 7 (Mutual Exclusion)**.** *If* `acq0` $\doteq 1$ *then* `acq1` $\doteq 0$.

We begin our proof sketch by recalling that, in Section 4.3, we assumed the read `[`x`]` in `acq0` read from the initialization. Here we will derive that fact from the assumption `acq0` $\doteq 1$.

Lemma 2 (Entered, Not Flagged)**.** *If* `acq0` $\doteq 1$ *then* `acq0`, `lrx` $\doteq 0$.

The derivation of Lemma 2 appears in Figure 4.10. Note, that some of the labels do not appear in Figure 4.7. In particular we have been referencing the memory accesses informally with sequences like `acq0,lrx`, but in the derivation we use the more precise `acq0`, l_1, `lrx`. Referenced rules appear in Figure 4.5 in Section 4.2.

In Figure 4.10 equation (4.1), the two let binding expressions and the assumed equality (`acq0` $\doteq$ 1) mean that the `if` expression must evaluate to 1. Then by if-alt, it is either the case that the `then` branch is equal to 1 and the condition expression (`valx`) is equal to 1 or the `else` branch is

equal to 1 and the condition is equal to 0. The `then` branch results in a contradiction (4.2), so we have that `valx` $\doteq 0$. Then it must be that the bound expression `lrx` is also equal to 0 by let-bind, in (4.3) as required. With this equality we can show that the read of `x` in `acq0` read from the initialization by using the value and write elimination to rule out the write `[`x`] := 1` in `acq1`.

With the read `[`x`]` connected to the initialization, we can use use the match `[`y`] @ acq1,lry` and the memory model level reasoning from Section 4.3 to derive the corresponding lemma for `acq1`.

Lemma 3 (Flagged, Failed to Enter)**.** *If* `acq1,lry` $\doteq 1$ *then* `acq1` $\doteq 0$.

Using similar reasoning to (4.3) we can show that the `if` condition must be 1 and that the `if` must be 0. By similar reasoning to (4.1) and (4.2) we can show that `acq1` $\doteq 0$, as required.

4.3.4 Reads to Memory

We will now formalize the reasoning of Section 4.3.2 and connect it to the expression level derivations of Section 4.3.3.

If we know that a read, `[x] @` L results in some value ($L_r \doteq n$) and read from a particular location ($L_r, l_{loc} \doteq l$), we also know that it read from some write ($\exists L_w, L_w \xrightarrow{\texttt{rf}} L_r$). The write is unknown but we consult a disjunction over a finite set of writes which could have satisfied the read for the location equal to L, l_{loc}. In the case of Dekker, we have two locations, x and y.

$$\texttt{[}x\texttt{]} := s \;@\; L_w \Rightarrow L_w = \texttt{init} \vee L_w = \texttt{acq1,lwx} \tag{4.4}$$

$$\texttt{[}y\texttt{]} := s \;@\; L_w \Rightarrow L_w = \texttt{init} \vee L_w = \texttt{acq0,lwy} \tag{4.5}$$

For our Dekker example, the writes in equations (4.4) and (4.5) are fixed because we are considering the whole program, so they do not require quantification over the label sequences to locate the writes. More often we will quantify over label sequences in the write-set definition (e.g. RingBuffer in Section 4.4).

We then perform write elimination by cases as illustrated previously to show that only the desired writes will satisfy the read. Once we have information about which writes are possible we can match the write value with the read value and continue our reasoning.

For each write we wish to eliminate, we use specified orders (vo-pushes) and axioms of the memory model (co-read) to prove a contradiction (co-cycle). The rule vo-pushes gives us a disjunction over visibility of the heads and tails of any two push orders. Recall that in Dekker we have push orders from the write to the read in each thread.

The rule co-read draws on the intuition we described earlier: Of all the writes that a read (L_r) is aware of ($L_1 \xrightarrow{\texttt{vo}} L_r$), the write that it reads from ($L_2 \xrightarrow{\texttt{rf}} L_r$) should happen last ($L_1 \xrightarrow{\texttt{co}} L_2$).

The rule co-cycl says that a write can't be coherence before itself. This is the contradiction we use to eliminate a write.

Recall diagrams 5 and 6 from Figure 4.9. Our goal is to show that a read of the initialization by, `[`y`]` in `acq1` results in a contradiction. Thus it could not have read 0. We will focus on the elimination in diagram 6.

Lemma 4 (Flag Read)**.** *if* $\texttt{init} \xrightarrow{\texttt{rf}} \texttt{acq0,lrx}$ *then* $\texttt{acq0,lwx} \xrightarrow{\texttt{rf}} \texttt{acq1,lry}$

By matching we have `[`y`]` @ `acq1,lry` and by let-left we have that $\exists n, \texttt{acq1,lry} \doteq n$. Then by reads-rf and the write set, we have a disjunction of two writes. We wish to eliminate the initialization. Thus we assume $\texttt{init} \xrightarrow{\texttt{rf}} \texttt{acq1,lry}$ and derive a contradiction.

The assumptions we require from Figure 4.9 correspond with the equations in Figure 4.11a. With them we can derive a cycle using the steps detailed in Figure 4.11b.

Note that the two sides of the disjunction in (4.10) are diagrams 5 and 6 in Figure 4.9 and the derivation is for the right side which corresponds with diagram 6. Thus, having considered both sides of the push disjunction we concluded that `acq1,lry` must have read from `acq0,lwy` by eliminating the initialization, as required.

4.3.5 Theorem 7

We can now complete the proof of Theorem 7. Assume $\texttt{acq0} \doteq 1$, then by Lemma 2 and the fact that only initialization writes the value 0 to `x`, we have that $\texttt{init} \xrightarrow{\texttt{rf}} \texttt{acq0,lry}$. Then, by Lemma 4 we have that `acq1,lrx` must have read from `acq0,lwy`. Since the value of written by `acq0,lwy` is 1, `acq1,lrx` must evaluate to 1, that is $\texttt{acq1,lrx} \doteq 1$. Finally, since we have $\texttt{acq1,lrx} \doteq 1$, we

$$\texttt{init} \xrightarrow{\texttt{rf}} \texttt{acq1,lry} \tag{4.6}$$

$$\texttt{init} \xrightarrow{\texttt{co}} \texttt{acq1,lwy} \tag{4.7}$$

$$\texttt{acq0,lwy} \xrightarrow{\texttt{push}} \texttt{acq0,lrx} \tag{4.8}$$

$$\texttt{acq1,lwx} \xrightarrow{\texttt{push}} \texttt{acq1,lry} \tag{4.9}$$

(a) Assumptions

$\texttt{acq1,lwx} \xrightarrow{\texttt{vo}} \texttt{acq0,lrx} \vee \texttt{acq0,lwy} \xrightarrow{\texttt{vo}} \texttt{acq1,lry}$	by vo-pushes, (4.8), and (4.9)	(4.10)
$\texttt{acq0,lwy} \xrightarrow{\texttt{vo}} \texttt{acq1,lry}$	by the right side of (4.10)	(4.11)
$\texttt{acq0,lwy} \xrightarrow{\texttt{co}} \texttt{init}$	by co-read, (4.6), and (4.11)	(4.12)
false	by co-cycle, (4.7), and (4.12)	

(b) Proof by Contradiction

Figure 4.11: Write Elimination Proof

employ Lemma 3 to show $\texttt{acq1} \doteq 0$. Thus, we have proven mutual exclusion for Dekker without using state. Our mechanized proof of Dekker is 260 lines in Coq without whitespace.

4.4 Induction on the Coherence Order: RingBuffer

In our logic, induction over the coherence order allows proofs to incorporate existing invariant based reasoning about single memory locations using the partial ordering of writes in the JOM. It also allows proofs to build on a structure for the surrounding program without explicitly defining it. When used in conjunction with quantification over label sequences, our logic can prove important properties of library algorithms like RingBuffer.

4.4.1 RingBuffer Algorithm and Specification

Our implementation, detailed in Figure 4.12, is a simplified form of the two-thread ring buffer that appears in the Linux kernel documentation [38]. It closely follows that of [82], though we

do not consider allocation and we use monotonic read and write indices [1]. It implements a queue using a fixed number of memory locations, which means that when the buffer is full `tryProd` will fail and when it is empty `tryCons` will fail.

We treat `wi`, `ri`, `b`, and `N` as fixed constants representing the write index offset (0), read index offset (1), buffer offset (2), and buffer size respectively. We also include the `succeed` and `fail` result constants for clarity.

Each procedure returns a value when the corresponding expression evaluates to a natural number. When the `tryProd` procedure successfully enqueues an element `x` in `q` it evaluates to `succeed`, otherwise it evaluates to `fail`. When the `tryCons` procedure successfully dequeues from `q`, it evaluates to the dequeued value, otherwise it evaluates to `fail`.

The writer index is managed by `tryProd` write `[wic] := w'` (green code/cell). It represents the tail of the queue. The reader index is managed by the `tryCons` write `[ric] := r + 1` (red code/cell). It represents the head of the queue. Both the reader and writer indices are allowed to increase indefinitely. When the buffer (blue cells) is empty the head and tail are equal (modulo `N`). When the buffer is full the writer index is one fewer than the reader index (both modulo `N`). The buffer index is the position at which a `tryProd` or `tryCons` enqueues or respectively dequeues an `x`. The buffer index is always calculated modulo the length of the buffer (`N`).

The correctness of the RingBuffer algorithm hinges on how the state of the writer index and reader index evolve over time. Specifically, we must show how the state of each evolves individually and then we must use that information to show how they evolve together.

As examples, the individual indices must progress by one index at a time. Otherwise, a `tryCons` invocation may skip an element in the queue or a `tryProd` invocation may leave an element that has already been dequeued for a `tryCons` to dequeue again. Together, the indices must stay within the bounds of the buffer size with respect to one another or the algorithm will exhibit similar problems.

We define the three theorems in our specification for RingBuffer. In our theorems we use

[1] This is worthy of special note since the logic of [82] employs ghost state with overflowing indices which is a non-trivial addition to the logic.

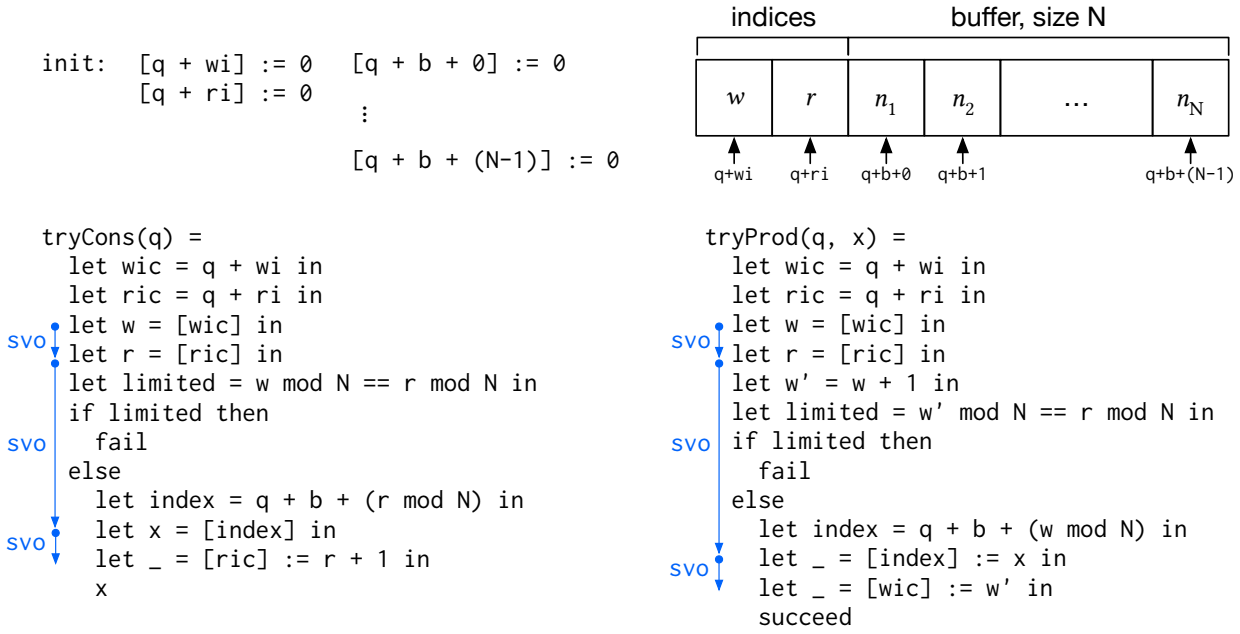


Figure 4.12: RingBuffer

the following conventions. We use $L_{\texttt{tryProd}}$ and $L_{\texttt{tryCons}}$ to represent arbitrary label sequences locating instances of their respective procedures. We use `ri`$*$, `wi`$*$, and `buff`$*$ to represent label sequences locating the read index, write index, and buffer location accesses within the procedures where $* \in \{\texttt{rd}, \texttt{wr}\}$ for reads and writes. Finally we use $L_1, \texttt{seq}$ to represent sequence concatenation.

Lemma 5 (Paired). *If we have* $L_{\texttt{tryCons}} \doteq \texttt{fail}$ *and* $L_{\texttt{tryProd}} \doteq \texttt{succeed}$ *then* $L_{\texttt{tryProd}}, \texttt{buffwr} \xrightarrow{\texttt{rf}} L_{\texttt{tryCons}}, \texttt{buffrd}$ *iff* $L_{\texttt{tryProd}}, \texttt{wird} \doteq L_{\texttt{tryCons}}, \texttt{rird}$

Informally, if a `tryCons` expression and `tryProd` expression both succeed, then the `tryCons` reads from the write to the buffer in the `tryProd` if and only if the reader and writer indices (resp.) used to calculate the buffer index are the same.

Lemma 5 is a natural correctness criteria for an unbounded linear queue. It guarantees that the n^{th} dequeue is paired with the n^{th} enqueue, where n^{th} is defined here by the monotonic reader and writer indices. As evidence, it's possible to use Lemma 5 to prove the key theorems for the concurrent queue of Herlihy and Wing [36] (see Appendix J).

Theorems 8 and 9 focus on the bounded nature of the queue. Intuitively, each dequeue (`tryCons`) should be paired with a newly enqueued value (`tryProd`) and not an old one, prevent-

ing stale values, and a dequeue (tryCons) musth have executed when more than one enqueue (tryProd) is attempted at a particular position in the buffer, preventing overwrites.

Theorem 8 (Produce). *If we have* $L_{\texttt{tryCons1}} \neq L_{\texttt{tryCons2}}$, $L_{\texttt{tryCons1}} \mathrel{\dot{\neq}} \texttt{fail}$, $L_{\texttt{tryCons2}} \mathrel{\dot{\neq}} \texttt{fail}$, $L_{\texttt{tryProd1}}, \texttt{buffwr} \xrightarrow{\texttt{rf}} L_{\texttt{tryCons1}}, \texttt{buffrd}$ *and* $L_{\texttt{tryProd2}}, \texttt{buffwr} \xrightarrow{\texttt{rf}} L_{\texttt{tryCons2}}, \texttt{buffrd}$ *then* $L_{\texttt{tryProd1}} \neq L_{\texttt{tryProd2}}$

Informally, If two distinct tryCons invocations succeed, then their corresponding tryProd invocations are distinct. That is, no two tryCons invocations can read from the same tryProd and thus no tryCons can read a stale value.

Theorem 9 (Consume). *If we have* $L_{\texttt{tryProd1}} \neq L_{\texttt{tryProd2}}$, $L_{\texttt{tryProd1}} \doteq \texttt{succeed}$, $L_{\texttt{tryProd2}} \doteq \texttt{succeed}$ *and* $L_{\texttt{tryProd1}}, \texttt{buffwr}, l_{loc} \doteq L_{\texttt{tryProd2}}, \texttt{buffwr}, l_{loc}$ *then there exists a* $L_{\texttt{tryCons}}$ *such that, if* $L_{\texttt{tryCons}} \mathrel{\dot{\neq}} \texttt{fail}$ *then* $L_{\texttt{tryProd1}}, \texttt{buffwr} \xrightarrow{\texttt{rf}} L_{\texttt{tryCons}}, \texttt{buffrd}$

Informally, If two distinct tryProd invocations succeed and write to the same place in the buffer, then there must be a tryCons invocation that reads from the write of the first tryProd invocation. That is, values in the buffer can't be overwritten without having been read. Note that we must assume the successful execution of the tryCons ($L_{\texttt{tryCons1}} \mathrel{\dot{\neq}} \texttt{fail}$) in our conclusion. This is a weakness of our abstraction over the expression semantics in dealing with the situation where the tryCons has not fully evaluated but the increment of the reader index inside the procedure executed. In the language of Herlihy and Wing, the tryCons is concurrent with the second tryProd but the update to the index has taken place allowing the second tryProd to proceed and reuse the buffer index.

Our mechanized the proofs of these theorems is approximately 3000 lines in Coq without whitespace. Notably, there is a large amount of duplicate proof code shared between tryCons and tryProd invariants. Here, we give a proof sketch that focuses on how our logic supports complex algorithms using induction on the partial coherence order. We will work through a series of lemmas building up to the proof of Theorems 8 and 9. We first give a simple example to show how the coherence order supports proofs of invariants and then examine a more complex example where it forms a scaffolding for write elimination.

4.4.2 Individual Invariants

In the process of proving Theorems 8 and 9 we will need to prove a series of lemmas. We begin with two invariants, one for the reader index and one for the writer index.

Lemma 6 (Monotonic Writer). *If we have* $\mathsf{writes}(L, q + wi, n_1)$, $L_{\texttt{tryProd}}, \texttt{wiwr}, l_{val} \doteq n_2$ *and* $L \xrightarrow{\texttt{co}} L_{\texttt{tryProd}}, \texttt{wiwr}$ *then* $n_1 < n_2$

Informally, if one write to the writer index is earlier than another then the first write's value is smaller than that of the second write. Here, $\mathsf{writes}(L, l, n)$ is used to encapsulate the equality of the location of the write expression, the equality of the value of the write expression, and the execution of the write expression itself. We use an abstract L because the first write may be the initialization for the writer index location, $q + wi$. The lemma for the reader index is similar.

The proofs for these lemmas proceed by induction on the coherence order of the writes to their respective indices. We will focus on `tryProd` and Lemma 6. The proof for `tryCons` is similar.

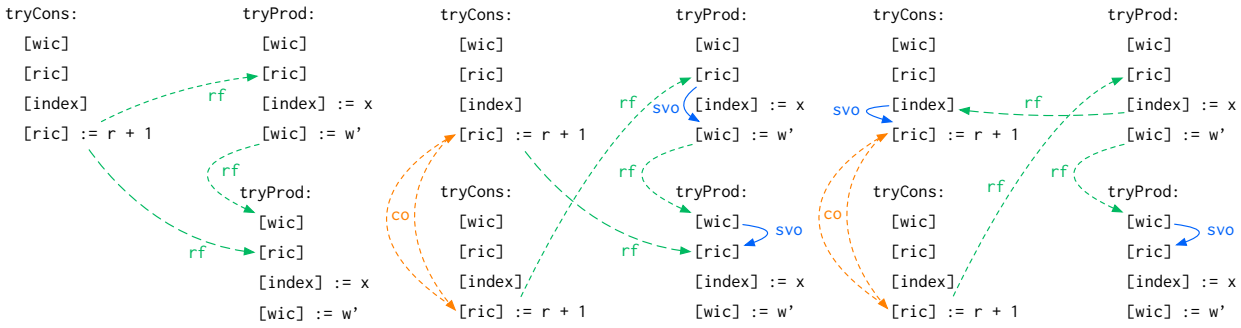
We will rely on the fact that every `tryProd` write to the writer index is an increment of the coi previous write. Intuitively, each successful addition to the buffer should increment the previous (by coi) writer index by 1.

We can perform induction on $L \xrightarrow{\texttt{co}} L_{\texttt{tryProd}}, \texttt{wiwr}$ (first assumption of the rule co-ind in Figure 4.6). Where $q + wi$ is the write index memory location, we wish to show:

$$P\ L\ L_{\texttt{tryProd}}, \texttt{wiwr} \triangleq \mathsf{writes}(L, q + wi, n_1) \Rightarrow L_{\texttt{tryProd}}, \texttt{wiwr}, l_{val} \doteq n_2 \Rightarrow n_1 < n_2$$

By co-ind, in the base case, we must show that $L \xrightarrow{\texttt{coi}} L_{\texttt{tryProd}}, \texttt{wiwr}$ (second assumption of co-ind). By the assumed lemma above we know the read in $L_{\texttt{tryProd}}$ will be the value written by L. Thus the `tryProd` will add one to the read value for its write, which is greater than the value written by L.

In the inductive case, we have some $L'_{\texttt{tryProd}}$ and we have that $L \xrightarrow{\texttt{co}} L_{\texttt{tryProd}}, \texttt{wiwr}$ implies that the value written by $L'_{\texttt{tryProd}}, \texttt{wiwr}$ is greater than the value written by L (third assumption of co-ind). As before, each `tryProd` reads from the the coi previous `tryProd` write and increments it. Thus the write in $L_{\texttt{tryProd}}$ is greater.



(a) Same `tryCons` (b) Backward read (c) Read from future write

Figure 4.13: Two-thread Invariant Cycles

4.4.3 Collective Invariants

Next we will establish bounds on each index relative to its counterpart in the opposite procedure. The writer index must not "wrap around" and pass the reader index otherwise previously enqueued items will be lost. Similarly the reader index must not pass the writer index otherwise already dequeued items will be dequeued again.

We write these bounds as invariants, here for `tryProd` in Lemmas 7. Notably, these are the same core invariants proved for RingBuffer by Turon et al in [82].

Lemma 7 (Writer Invariant). *If we have* $L_{\texttt{tryProd}} \doteq \texttt{success}$, $L_{\texttt{tryProd}}, \texttt{wiwr}, l_{val} \doteq w$ *and* $L_{\texttt{tryProd}}, \texttt{rird} \doteq r$ *then* $w < r + N$

Informally, if a `tryProd` succeeds and writes to the writer index, then the value was smaller than the reader index it saw plus the size of the buffer, N. The lemma for the reader invariant is similar but shows that $r \leq w$.

We will again focus on `tryProd` and Lemma 7. The proof for `tryCons` is similar. We will show $w < r + N$. The proof proceeds by induction on $\texttt{init} \xrightarrow{\texttt{co}} L_{\texttt{tryProd}}, \texttt{wiwr}$.

In the base case, we know that $\texttt{init} \xrightarrow{\texttt{coi}} L_{\texttt{tryProd}}, \texttt{wiwr}$ and we know that the `tryProd` at $L_{\texttt{tryProd}}$ will read from the initialization so the invariant will be satisfied no matter which write the value for r came from.

In the inductive case, we know that the writer index value the `tryProd` reads has the desired property by coi. We also know that it must have come from the most recent previous `tryProd`

and is consequently one fewer than w. That is, we can show $w - 1 < r' + N$, where r' represents the reader index value seen by the previous `tryProd`. From this we derive, $w \leq r' + N$. Then since we wish to show $w < r + N$, it's enough to show that the reader index value of the later `tryProd` must be larger than the value seen by the coi previous `tryProd`. That is, we will show $r' < r$. With the scaffolding from induction in place we can perform write elimination using the ordered `tryProd`s.

If both `tryProd` invocations read from the same write, see Figure 4.13a, then $r' = r$, and we consider cases for, $w \leq r + N$, with r substituted for r'. If $w < r + N$, then we are done. Otherwise $w = r + N$ and $w \bmod N = (r + N) \bmod N$ which means the buffer is full and we can show that the second `tryProd` would have taken the first branch of the `if` and could not have written to the writer index, a contradiction.

That leaves distinct writes. We consider cases of the total coherence ordering of the two `tryCons` writes to the reader index that were read by the `tryProd`s. If the write of r' is coherence order earlier than the write of r we apply reader index monotonicity to show $r' < r$ and we are done. If the second `tryProd` invocation read from an earlier `tryCons`, depicted in Figure 4.13b as a red dashed rf edge, it would imply that $r < r'$. We will show a contradiction.

Recall the visibility orders of the algorithm definition in Figure 4.12. Also recall that the two `tryProd`s are related by a read of the writer index. Intuitively, we use the visibility orders to ensure that the second `tryProd` is aware of the second `tryCons` through the first `tryProd`'s read. As a result the first `tryProd` cannot ignore the second `tryCons` in favor of a coherence order earlier write. Said another way, the second `tryProd` would have to read back into the past to see a state where r' is less than r. This results in the cyclic coherence order edges between the two writes to the reader index as shown in Figure 4.13b and in turn, a contradiction. Thus we have shown that $r' < r$.

4.4.4 Lemma 5 and Theorems 8 and 9

Now we can complete the proof of Lemma 5 by unifying the invariants for `tryProd` and `tryCons`, We will show that when a `tryCons` invocation reads the buffer at a particular index written by a

companion tryProd invocation, the reader and writer indices used to calculate the index are the same. This proof brings together the invariants of Lemmas 7 and its tryCons counterpart.

By the semantics of reads, if a write and read of a buffer memory location are associated then the calculated index used to determine the memory location in the associated tryProd and tryCons invocations must be the same. Let the reader index value used in the tryCons calculation be r and the writer index value used in the tryProd calculation be w. If the reader and writer indices are not the same, then by the calculations on lines 10 of tryCons and 11 of tryProd we know that, $r = w + x \times N$, for some integer $x \neq 0$.

We consider the cases for x, first $x < 0$. Then $r = w + x \times N$ implies $r \leq w - N$. We know that if the tryProd containing the write to the buffer executed completely then w and its own r' are related according to Lemma 7 with $w < r' + N$. Then we have $w - N < r'$, and by assumption we have $r \leq w - N$, which gives us $r \leq w - N < r'$ and $r < r'$.

Figure 4.13c illustrates the contradiction in these two reads. By reader index monotonicity and because $r < r'$, wherever the tryProd got its reader index r' is coherence order after the tryCons that computed its index location using r. This is the co edge pointed downward. But, we can establish, through the read of the buffer and the visibility orders, that the coherence order later write happened-before the coherence order earlier write and thereby derive a contradiction. Intuitively, we ensure the visibility of the "future" write through the specified visibility orders. That is, through the visibility orders, the read learns about a write that has yet to take place.

The case for $x > 0$ is similar but uses the tryCons invariant to prove that the tryCons must have seen a writer index coherence order after another tryProd write to the same buffer index. In turn, this implies that it would have ignored the new state of that index to read into the past.

With Lemma 5 in hand we can prove the two main theorems. Theorem 8 follows from the fact that the two executed tryCons procedures must have read different reader index values. Then by Lemma 5 their paired tryProds must also write distinct writer indices and thus be in coherence order. Theorem 9 follows from two facts. First, the two tryProds must have used writer indices that are equal modulo N. Second, the later tryProd must have seen a reader index that was greater than its writer index less the buffer length, $w - N < r$. Since r is larger than $w - N$ there

must be some `tryCons` for any $r \leq w - N$ and by Lemma 5 it must have read from the earlier `tryProd`'s index write.

4.5 Summary

Here, we have presented a sound, stateless logic for reasoning about the correctness of lock free concurrent algorithms executing on the JOM. As examples, we proved the correctness of Dekker and RingBuffer. The result is that these algorithms, when paired with their attendant specified orders and given to our compiler can produce fast, correct code for many memory models.

CHAPTER 5

Related Work

For a long time, researchers have known that correctness can depend critically on the execution order of two instructions. Fences are a crude way of ensuring that two instructions execute in order. Kuperstein, Vechev, and Yahav [44] used a notion of specified orders as the output of a synthesis algorithm. Like us, they see these orders as part of a correct program, but inferred from a correctness property, rather than specified. The idea of specified orders appeared for first time in publications in 2014–2015, namely in the 2014 PhD dissertation of [54], and in the POPL 2015 paper by [19]. Crary and Sullivan's POPL 2015 paper introduced the RMC memory model together with a semantic foundation that includes specified orders. More recently a "Placed Before" intra-thread ordering relation was proposed for the C++ concurrency standard [59]. It captures the key idea of the visibility ordering (specifying ordering dependencies explicitly) but with a focus on ruling out thin-air reads.

Beyond the concept of specified orders we will consider work related to this thesis in three categories: fence insertion, memory models and semantics, and verification for weak memory models.

5.1 Fence Insertion

Many authors have presented approaches to insert fences that enforce sequential consistency, including Lee et al. [53], Fang et al. [29], and Alglave et al. [4]. An alternative to programmer specification of orders is *inference* of orders. The idea of inference is somewhat different from *type inference*, which can be understood as articulating program invariants. In the case of order inference, the challenge is to articulate assumptions needed to prove a correctness property.

Kuperstein, Vechev, and Yahav [44], presented promising work on inference; they infer specified orders from a program, a correctness property, and a memory model. Their approach first runs a whole-program state-space exploration algorithm that produces a logical formula, then solves the formula to get a set of specified orders, and finally uses those orders to insert fences. Their approach to enforce an order (i_1, i_2) is to insert a fence right after i_1 or right before i_2. The whole-program nature of their approach means that while the inserted fences are sound in the given context, they may be unsound in a different context. Still, their approach can give worthwhile feedback to an algorithm designer who tries to specify a set of orders that are sufficient to prove correctness. We note that our choice of correctness property (opacity) of transactional transactional memory algorithms is currently beyond the capabilities of the Kuperstein-Vechev-Yahav approach. What is needed here is a more powerful language for specifying correctness properties along with a suitable generalization of the approach. This can be an exciting direction for future work.

Kuperstein et al. [45], Meshman et al. [63], and Dan et al. [21] have presented approaches to a related inference problem, allowing degrees of infinite-state programs, but seemingly without specified orders as an intermediate step towards fences. Their approaches can likely be recast as inference of specified orders. Again, a direction for future work is to make their specification languages more powerful to enable specification of correctness of TM algorithms.

Liu et al. [57] presented an execution-based approach to inference, in which they run the program on a memory model and then use the traces to infer fences. This technique can likely be recast to infer specified orders instead.

Specified orders are restrictions on the possible executions of a program. In this way they are similar to previous work on using annotations to restrict scheduling for correctness [20] and for testing [40]. Our work differs in its focus on the restriction of the possible executions due to instruction reordering (or the appearance thereof) as opposed to the restriction of the possible executions due to thread scheduling.

5.2 Memory Models

The original Java Memory Model [60] included an early attempt to model standard compiler optimizations while ruling out thin-air reads. The specification in that work was part prose and part formal definitions and the "causality" mechanisms at the heart of the model made the definitions complex. Taken together these issues made the model unsuitable for the tasks which one normally formalizes a programming language, namely accurate discourse, metatheory, and algorithm verification. Later work by [77] manually examined a large suite of litmus tests to show that the model disallowed some standard compiler optimizations. Eventually the model was fully formalized in Coq by [39] and [7], but, to the best of our knowledge, there is no way to easily test the behavior of example programs. By contrast we have constructed a readable and testable model for Java's new access modes.

The C11 memory model, which served as the inspiration for Java'a access modes, has seen extensive study. It was originally formalized by [8] with later revisions to include read-modify-writes and fences [76]. The work of [85], from which we draw the largest set of our C11 litmus tests, studied the soundness of common compiler optimizations under the model of C11 by [65]. Our monotonicity theorem is modeled after the same theorem from [85]. Most recently, the axiomatic model of [47] incorporated the proposed fixes of [85] and addressed the unsound compilation strategies which we discussed in Section 3.5. Also, further progress has been made on new models for C11 that more successfully support standard compiler optimizations without the problem of thin-air reads [43, 71]. However, like the original Java memory model, these models rely on complex formal constructs (promises and event structures respectively). Again, our semantics remains relatively simple thanks to the JAM's broad definition of causal cycles.

As outlined previously there are several important differences between C11 and Java. First, the partial coherence order required careful consideration. The consequences of this design choice manifests most clearly where atomic read-writes are concerened. Second, the lack of legacy features, like C11's release sequences, and the simple mechanism that forbids causal cycles allowed us to build a simple model. In turn, the simplicity of the model makes it more readable than existing C11 models and it allowed us to argue forcefully that the model should adopt a total

coherence order.

These differences appear most clearly in our litmus test comparison with RC11 model of [47]. Of particular note are the `mp_relacq_rs` and `lb` tests. In the first case the JAM is weaker than C11 because it does not support release sequences, in the second case it is stronger because the C11 specification gives no concrete definition for how to rule out thin-air reads. Notably, the RC11 herd model also forbids causal cycles in `po | rf` so the load buffering behavior is forbidden. This method of preventing thin-air reads in RC11 is included to enable their proofs of soundness for compilation to POWER and not as a representation of the C11 specification.

Programs that mix SC/volatile access modes are the primary focus of [47]. In particular the leading and trailing fence insertion approach to compilation was shown to be unsound for Power under models prior to RC11. We compared our model with RC11 in Section 3.5. Our semantics correctly forbids the behavior described in `IRIW-sc-rlx-acq`, `IRIW-aqc-sc` and `Z6.U` in keeping with the JAM documentation.

Hardware memory models have also seen extensive study. x86 was studied by [69] and [5]. We use the model included with Herd in our litmus test comparisons and we saw that x86 was stronger than the JAM, as expected. ARM processors have traditionally had a much weaker memory model when compared with x86. Recently the model for ARMv8 [74] expanded the guarantees made by the architecture to include multi-copy-atomicity. We saw the effects of this in the behavior of the `IRIW-*` and `WRC-*` litmus tests from our comparison betwen the JAM and ARMv8. As expected the JAM is weaker than ARMv8 in every case except where cycles in `po | rf` are concerened.

Our mechanized semantics is based on the history fragment of the Relaxed Memory Calculus of [19]. We use their concept of specified push orders and we draw inspiration for our definitions from their notion of visibility. Also, the proof of our theorems benefited greatly from the library of lemmas included with the RMC mechanization. However, their purpose was to model a weaker version of C11 in the interest of generality while, our goal is to model the JAM. Importantly, we do not employ the execution orders of RMC, our coherence definition is far more compact and we have added the `corr` rule to follow a more standard notion of coherence.

5.3 Verification

In the presence of a very weak-memory model like the JAM-, a key problem for logics that reason with state is ensuring a consistent ordering of writes in memory, across threads. The best work on capturing write orderings using state is the work of Turon et al. [82]. They combine separation logic, ghost variables and protocols in a powerful logic for verifying algorithms running on the release/acquire fragment of C11. GPS has been used to verify many algorithms including the RingBuffer that we have presented and the linux RCU algorithm [81]. Our specification for Ring Buffer is stronger, in that Lemma 5 can be used to show that reordering within the buffer is impossible. On the other hand, GPS leverages ghost state to do the proof with integer values that wrap (overflow) which our logic cannot do. We also do not handle allocation.

The protocols of GPS, which are, in the words of authors, the lifeblood of prior concurrency logics require a total order on writes to the same location in their proofs of soundness. Moreover, even assuming a total order on writes, the protocols of GPS cannot be used to prove correctness of some algorithms. As an example consider Peterson's lock.

In Peteron's lock, when a thread examines the state of the victim field, the information it learns is not enough to tell whether the other thread's ordering for the two writes to the victim field is the same. This is important because the ordering of the victim writes determines which thread will enter the critical section when they compete. For example, if the first thread sees the second thread's victim write after its own it must have some way to know that the other thread sees the same ordering of writes. Otherwise, the the second thread could see the reverse ordering, read the first threads victim value, and also enter the critical section.

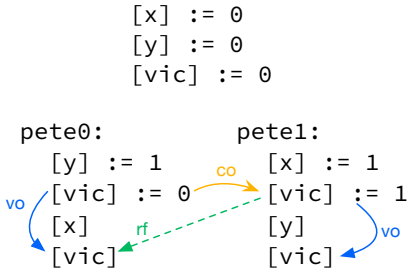


Figure 5.1: Peterson Victim

The protocols of GPS are based on partial orders but the victim state can't be encoded as a partial order since it could progress from 0 to 1 or vice versa as both orderings of the writes are possible in an execution.

By contrast, in Figure 5.1 we have a JOM execution with two competing lock procedures of

Peterson. The fact that `pete0` reads from the victim write of `pete1` and knows about its own victim write (gray dashed edges) means that the two are coherence-ordered in memory by co-read (black edge). If `pete1` were to mirror `pete0` and read from `[vic] := 0` it would result in the an opposing coherence order between the two writes, which would be a cycle and a contradiction.

The work on GPS is one part of a large body of impressive work on concurrent separation logics for weak memory that includes it's forebearers [86], [84], and [83]. These logics are based on the foundational work of [37] (TTPP), [70] (interference), [42] (rely/guarantee) and [67] (CSL).

More recently [80] have constructed a logic for the promising semantics of [43] which features a version of the C11 semantics with fully relaxed memory accesses and without thin-air reads. Though, in that work, they do not prove correctness of any algorithms. Also of note is the recent work on the Fenced Separation Logic of [28] which is an extension of RSL. The soundness of FSL requires a restriction of the C11 memory model which prevents read-write reorderings by a compiler which is one possibly way to satisfy acyclic(po | rf).

The work of Alglave and Cousot [2] aims to enable proofs for many memory models. In their proof method they specify a set of "bad reads" that will cause the algorithm to fail and prove correctness under the absence of those reads. Then a target memory model must rule out those reads. In the presence of a very weak memory model that set of bad reads must be small or empty, placing the burden on the algorithm to forbid such reads by using fences or other synchronization methods. The result would be either, a performance penalty when executing the algorithm on strong memory models that already forbid those reads or the work of extra proofs and implementations for those stronger memory models. By contrast, our goal is to construct proofs for the JOM. However, it is worth noting that the addition of specified orders means that our compiler can take our specified orders and provide fast, correct executable code for any stronger target memory model.

Outside the context of weak memory, researchers leveraged the early work of [67] to great effect. The works of [31] and [33] supported dynamically allocated and re-entrant locks respectively. Message passing has been considered in [87] and [10]. Fork and join support appeared in [27]. Finally, in [56] they construct a logic to prove liveness properties for concurrent objects

building on their work in [55].

Thin-air reads invalidate basic thread-local state based reasoning as noted in [86]. The semantics of the JOM does not permit thin-air reads by requiring that all executions satisfy acyclic(po | rf), but our approach to reasoning does not rely on that guarantee and our logic is sound in the presence of thin-air reads.

Instead, write elimination starts from an over-approximation of possible writes to a memory location. In our proofs we make no attempt to rule out writes based on data or control dependencies and so we would include the, intuitively impossible, writes that appear in the classic thin-air read examples, e.g. Figure 3.2a.

To address such cycles we would require specified visibility orders (vo) to augment the happens-before relationship to rule out such writes by deriving cycles in the coherence order. As a result, we may need extraneous visibility orders to rules out writes that, in practice, can never satisfy a read. Thus, our reasoning principles remain sound but in some cases performance may suffer when the orders are compiled to synchronization.

The effect of the extra visibility orders would certainly be obviated by the Java compiler's own implementation of acyclic(po | rf), but the specific approach would affect the semantics. In recent work Ou et al. [68] showed that a compiler targeted at annotated accesses, like the JOM's `VarHandle` API, can eliminate thin-air reads with a worst case overhead of 6.3% for many data structures by forbidding read-write reordering. Adopting this approach would mean extending the definition of vo in our semantics to include all read-write pairs in program order.

Importantly, the ability to reason in the presence of thin-air reads differentiates our logic from recent work by Raad et al. which is also focused on concurrent libraries for weak memory models [75]. On the other hand, their framework is more general with respect to the memory model as they can simply specify it as another library. Further, their framework can handle libraries without obvious linearization points which we do not address.

CHAPTER 6

Conclusion

We have demonstrated that the correctness of lock-free concurrent algorithms can be proved once for implementations that can be compiled to run correctly and efficiently on all mainstream memory models. We have accomplished this by constructing a fence insertion algorithm and compiler, the first memory model for Java's Access Modes, and a mechanized logic.

In future work we hope to have our memory model adopted as part of the Java language specification. We also hope to pursue the verification of the algorithms in Java's concurrent utility library. Finally, we see specified orders as an answer to the issues around mixed mode access semantics in Java's Access Modes and we are interested in studying their characteristics as synchronization primitives.

[1] GNU GCC 4.8.2. Built-in functions for atomic memory access. https://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/_005f_005fsync-Builtins.html#g_t_005f_005fsync-Builtins, 2013. [Online, accessed Feb 2015].

[2] Jade Alglave and Patrick Cousot. Ogre and pythia: An invariance proof method for weak consistency models. *SIGPLAN Not.*, 52(1):3–18, January 2017.

[3] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, DAMP '09, pages 13–24, New York, NY, USA, 2008. ACM.

[4] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.*, 39(2):6:1–6:38, May 2017.

[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014.

[6] ARM. Arm compiler toolchain assembler reference. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489c/CIHGHHIE.html, 2011. [Online, accessed Feb 2015].

[7] David Aspinall and Jaroslav Ševčík. Formalising Java's data race free guarantee. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'07, pages 22–37, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011.

[9] Mark Batty and Peter Sewell. The thin-air problem. http://www.cl.cam.ac.uk/~pes20/cpp/notes42.html, 2014. [Online, accessed Dec 2018].

[10] Christian J. Bell, Andrew W. Appel, and David Walker. Concurrent separation logic for pipelined parallelization. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 151–166, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] John Bender. Parry. https://bitbucket.org/ucla-pls/parry, 2015.

[12] Garrett Birkhoff. *Lattice Theory*. American Mathematical Society, New York, NY, 1948.

[13] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[14] Joseph Cheriyan, Howard Karloff, and Yuval Rabani. Approximating directed multicuts. In *FOCS*, pages 320–328, 2001.

[15] C++ Standards Committee, Pete Becker, et al. Programming languages-c++(final committee draft). c++ standards committee paper wg21/n3092= j16/10-0082, 2010.

[16] Jon Corbet. Access_once(). `https://lwn.net/Articles/508991/`, 2012. [Online, accessed Dec 2018].

[17] Intel Corp. Intel® 64 and ia-32 architectures software developer's manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer- manual-325462.pdf`, 2015. [Online, accessed Feb 2015].

[18] Marie-Christine Costa, Lucas Létocart, and Frédéric Roupin. Minimal Multicut and Maximal Integer Multiflow: A Survey. *European Journal of Operational Research*, 162:55–69, 2005.

[19] Karl Crary and Michael Sullivan. A calculus for relaxed memory. In *Proceedings of POPL'15, ACM Symposium on Principles of Programming Languages*, 2015.

[20] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405, New York, NY, USA, 2013. ACM.

[21] Andrei Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In *SAS*, 2013.

[22] Chi Cao Minh Dave Dice, Nir Shavit. Tl2 and tl2 eager. `https://bitbucket.org/ucla-pls/stamp-tl2-x86/src/master/tl2.c`, 2015. [Online, accessed Feb 2015].

[23] Tiago de Paula Peixoto. Graph-tool, efficient network analysis. `https://graph-tool.skewed.de/`, 2015. [Online, accessed Feb 2015].

[24] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Distributed Computing*, pages 194–208. Springer, 2006.

[25] Dave Dice and Nir Shavit. Tlrw: Return of the read-write lock. In *SPAA*, pages 284–293, New York, NY, USA, 2010. ACM.

[26] David Dice. A race in locksupport park() arising from weak memory models. `https://blogs.oracle.com/dave/entry/a_race_in_locksupport_park`, 2009. [Online, accessed Feb 2015].

[27] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP'09)*, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with fsl++. In Hongseok Yang, editor, *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017*, pages 448–475. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[29] Xing Fang, Jaejin Lee, and Samuel Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*, 2003.

[30] GNU. Glpk (gnu linear programming kit). `https://www.gnu.org/software/glpk/`, 2015. [Online, accessed Feb 2015].

[31] Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. *Electron. Notes Theor. Comput. Sci.*, 276:171–190, September 2011.

[32] Richard Grisenthwaite. Barrier litmus tests and cookbook. `http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pd`, 2009. [Online, accessed Feb 2015].

[33] Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about java's reentrant locks. In G. Ramalingam, editor, *Programming Languages and Systems*, pages 171–187, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[34] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[35] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[36] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[37] C. A. R. Hoare. Towards a theory of parallel programming. In Per Brinch Hansen, editor, *The Origin of Concurrent Programming*, pages 231–244. Springer-Verlag New York, Inc., New York, NY, USA, 1972.

[38] David Howells and Paul E. McKenney. Circular buffers. `https://www.kernel.org/doc/Documentation/circular-buffers.txt`, 2017. [Online, accessed July 2017].

[39] Marieke Huisman and Gustavo Petri. The Java memory model: a formal explanation. *VAMP*, 7:81–96, 2007.

[40] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Rosu, and Darko Marinov. Improved multithreaded unit testing. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 223–233, New York, NY, USA, 2011. ACM.

[41] JDK9. Varhandle (java se 9 & jdk 9). `https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html`, 2017. [Online, accessed March 2019].

[42] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[43] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of POPL'17, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 2017.

[44] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 111–120, Austin, TX, 2010.

[45] Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, 2011.

[46] Kutzi. Answer to: What is the most frequent concurrency issue you've encountered in java? `https://stackoverflow.com/a/462648`, 2018. [Online, accessed May 2019].

[47] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, 2017. ACM.

[48] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[49] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[50] Doug Lea. The jsr-133 cookbook for compiler writers. `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf`, 2004. [Online, accessed July 2017].

[51] Doug Lea. Jep 193. `http://openjdk.java.net/jeps/193`, 2017. [Online, accessed March 2019].

[52] Doug Lea. Using jdk 9 memory order modes. `http://gee.cs.oswego.edu/dl/html/j9mm.html`, 2018. [Online, accessed March 2019].

[53] Jaejin Lee and David Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*, 50(8), August 2001.

[54] Mohsen Lesani. *On the Correctness of Transactional Memory Algorithms.* PhD thesis, UCLA, 2014.

[55] Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 459–470, New York, NY, USA, 2013. ACM.

[56] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 385–399, New York, NY, USA, 2016. ACM.

[57] Feng Liu, Nayden Nedeve, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, 2012.

[58] Lun Liu, Todd Millstein, and Madanlal Musuvathi. A volatile-by-default jvm for server applications. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):49, 2017.

[59] Daniel Lustig. P1239r0 - placed before. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1239r0.html`, 2018. [Online, accessed Dec 2018].

[60] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[61] Virendra J Marathe, Michael F Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N Scherer III, and Michael L Scott. Lowering the overhead of nonblocking software transactional memory. Technical Report 893, University of Rochester, 2006.

[62] Daniel Marino, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. The silently shifting semicolon. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[63] Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. Synthesis of memory fences via refinement propagation. In *Proceedings on Static Analysis Symposium, Lecture Notes in Computer Science Volume 8723*, pages 237–252, 2014.

[64] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008. [`http://stamp.stanford.edu`, online, accessed Nov 2014].

[65] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model. In *ACM SIGPLAN Notices*, volume 48, pages 187–196. ACM, 2013.

[66] University of Rochester and Lehigh University Departments of Computer Science. Rstm byteeager. `https://code.google.com/p/rstm/source/browse/trunk/libstm/algs/byteeager.cpp`, 2015. [Online, accessed Feb 2015].

[67] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.

[68] Peizhao Ou and Brian Demsky. Towards understanding the costs of avoiding out-of-thin-air results. *Proc. ACM Program. Lang.*, 2(OOPSLA):136:1–136:29, October 2018.

[69] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-tso. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[70] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6(4):319–340, December 1976.

[71] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. *SIGPLAN Not.*, 51(1):622–633, January 2016.

[72] LLVM Project. Llvm language reference manual. `http://llvm.org/docs/LangRef.html`, 2015. [Online, accessed Feb 2015].

[73] Sage Project. Sagemath. `http://www.sagemath.org/`, 2015. [Online, accessed Feb 2015].

[74] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, December 2017.

[75] Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.*, 3(POPL):68:1–68:31, January 2019.

[76] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising c/c++ and power. *Acm Sigplan Notices*, 47(6):311–322, 2012.

[77] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*, pages 27–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[78] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.

[79] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[80] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. A separation logic for a promising semantics. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 357–384, Cham, 2018. Springer International Publishing.

[81] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *PLDI*, 2015.

[82] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. Gps: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of OOPSLA'14, Object-Oriented Programming Systems, Languages and Applications*, 2014.

[83] Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[84] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, 276:335–351, September 2011.

[85] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the c11 memory model and what we can do about it. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 209–220, New York, NY, USA, 2015. ACM.

[86] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for c11 concurrency. In *Proceedings of OOPSLA'13, Object-Oriented Programming Systems, Languages and Applications*, 2013.

[87] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 194–209, Berlin, Heidelberg, 2009. Springer-Verlag.

[88] John Wickerson and Mark Batty. Taming the complexities of the c11 and opencl memory models. *arXiv preprint arXiv:1503.07073*, 2015.

APPENDIX A

Classic Concurrent Algorithms and Specified Orders

In this appendix we include the source code for each of the "classic" algorithms that was used in our fence insertion experiments. Each implementation comes from the Musketeer benchmarks and is reproduced here line-for-line. This is intended to serve as a reference for the line numbers detailed in the fence placement results tables in the main paper.

A.1 Dekker's Mutex

In Figure A.1 we have the two procedures used to simulate process interaction in the Musketeer Dekker implementation. Note that `flag1`, `flag2` and `turn` are shared. Recall that the orders we have defined are store-load orders between lines 2 and 3, 7 and 3 in bothe procedures. Note that the orders along back edges from the stores to `flag1` and `flag2` at the end of the `while` loops are required for the same reason as the order between the first stores to `flag1` and `flag2` and the loads just below them.

A.2 Parker Mutex

In Figure A.2 we have the procedure which contains the bug in the Parker implementation in the JVM. Note that `_counter` is shared. Recall that the order we have defined is a store-any order between lines 3 and 5.

```c
void* thr1(void * arg) {
  flag1 = 1;
  while (flag2 >= 1) {
    if (turn != 0) {
      flag1 = 0;
      while (turn != 0) {};
      flag1 = 1;
    }
  }
  // begin: critical section
  //x = 0;
  //assert(x<=0);
  // end: critical section
  turn = 1;
  flag1 = 0;
}
```

```c
void thr2(arg){
  flag2 = 1;
  while(flag1 >= 1) {
    if(turn != 1) {
      flag2 = 0;
      while(turn != 1){}
      flag2 = 1;
    }
  }
  // begin: critical section
  //x = 1;
  //assert(x>=1);
  // end: critical section
  turn = 1;
  flag2 = 0;
}
```

$$O_{\text{Dekker}} = \{\ \mathtt{W}(\text{flag1}) \xrightarrow{\text{push}} \mathtt{R}(\text{flag2}), \mathtt{W}(\text{flag2}) \xrightarrow{\text{push}} \mathtt{R}(\text{flag1})\ \}$$

Figure A.1: Dekker's Mutex

```
1  void park() {
2    if (_counter > 0) {
3      _counter = 0;
4      // mfence needed here
5      return;
6    }
7    if (mutex_trylock(&__unbuffered_mutex) != 0) return;
8    if (_counter > 0) { // no wait needed
9      _counter = 0;
10     mutex_unlock(__unbuffered_mutex);
11     return;
12   }
13   __unbuffered_did_park=1;
14   cond_wait(__unbuffered_cond, __unbuffered_mutex);
15   _counter = 0;
16   mutex_unlock(__unbuffered_mutex);
17 }
```

$$O_{\text{Parker}} = \{\ \mathtt{W}(\text{counter}) \xrightarrow{\text{push}} *\}$$

Figure A.2: Parker

```
1  void* thr1(void * arg) {
2    flag1 = 1;
3    turn = 1;
4    while (flag2==1 && turn==1);
5    // begin: critical section
6    // end: critical section
7    flag1 = 0;
8  }
```

```
1  void* thr2(void * arg) {
2    flag2 = 1;
3    turn = 0;
4    while (flag1==1 && turn==0);
5    // begin: critical section
6    // end: critical section
7    flag2 = 0;
8  }
```

$$O_{\text{Peterson}} = \{\ \mathtt{W(flag1)} \xrightarrow{\text{push}} \mathtt{R(flag2)}, \mathtt{W(flag2)} \xrightarrow{\text{push}} \mathtt{R(flag1)}\ \}$$

Figure A.3: Peterson's Mutex

A.3 Peterson's Mutex

In Figure A.3 we have the two procedures used to simulate process interaction in the Musketeer Peterson implementation. Note that `flag1`, `flag2` and `turn` are shared. Recall that the orders we have defined are both store-load orders between lines 5 and 7 and also between lines 14 and 16.

A.4 Lamport's Mutex

Here we detail two example executions for Lamport's mutual exclusion algorithm. They illustrate the need for at least two orders in the implementation included in Musketeer's "classic" benchmarks. The relevant source code appears in Figure A.4. Note that `x` and `y` are shared. The first order is between the stores to `x` on line 4 and the loads of `y` on line 5. The second order is between the stores to `y` on line 10 and the loads of `x` on line 11.

For the example executions in Figures A.5 and A.6 we use $\mathtt{R(y)} : 0$ to denote a 0 valued result loaded from the address represented by `y`, $\mathtt{W(y, 1)}$ to denote a store of the value 1 to the same address, and *enter* to denote the point at which a process enters the critical section. On the right margin we note where the loads correspond with `if` statements.

To see that the first order is necessary consider the execution in Figure A.5, where only the

```c
void thr1() {
 L0:
  b1 = 1;
  x = 1;
  if (y != 0) {
    b1 = 0;
    goto L0;
  }

  y = 1;
  if (x != 1) {
    b1 = 0;

    if (y != 1) {
      goto L0;
    }
  }
  // begin
  // end
  y = 0;
  b1 = 0;
}
```

```c
void thr2() {
 L1:
  b2 = 1;
  x = 2;
  if (y != 0) {
    b2 = 0;
    goto L1;
  }

  y = 2;
  if (x != 2) {
    b2 = 0;

    if (y != 2) {
      goto L1;
    }
  }
  // begin
  // end
  y = 0;
  b2 = 0;
}
```

$$O_{\text{Lamport}} = \{\ \mathtt{W(x)} \xrightarrow{\text{push}} \mathtt{R(y)}, \mathtt{W(y)} \xrightarrow{\text{push}} \mathtt{R(x)}\ \}$$

Figure A.4: Lamport's Mutex

thr1	thr2 :	
$\mathsf{R}(y, 0)$		$\texttt{if} : y \neq 0$
	$\mathsf{R}(y, 0)$	$\texttt{if} : y \neq 0$
$\mathsf{W}(x, 1)$		
$\mathsf{W}(y, 1)$		
$\mathsf{R}(x, 1)$		$\texttt{if} : x \neq 1$
enter		
	$\mathsf{W}(x, 2)$	
	$\mathsf{W}(y, 2)$	
	$\mathsf{R}(x, 2)$	$\texttt{if} : x \neq 2$
	enter	

Figure A.5: Lamport's Mutex, Bad Execution 1

stores to x move past the guards that check if y is equal to 0.

Separately, if the stores to y are allowed to pass the `if` statements that check that value of x, the execution in Figure A.6 is possible.

thr1	thr2 :	
W(x, 1)		
R(y, 0)		if : $y \neq 0$
R(x, 1)		if : $x \neq 1$
	W(x, 2)	
	R(y, 0)	if : $y \neq 0$
	W(y, 2)	
	R(x, 2)	if : $x \neq 2$
	enter	
W(y, 1)		
enter		

Figure A.6: Lamport's Mutex, Bad Execution 2

APPENDIX B

Fence Insertion Algorithm Correctness Proof

The lift function. We first define the helper function lift that later will enable us to state some properties succinctly. Let $G = (V, E, \ell)$ and let $K \subseteq E$, and suppose we have i_1, i_2 such that $p \in \mathsf{paths}(\ \mathsf{Refine}(G, K), i_1, i_2)$. We define $\mathsf{lift}(p)$ to be the corresponding path in G, that is, the path that for each pair of edges $(j_1, v_{j_1,j_2}), (v_{j_1,j_2}, j_2)$ in p instead has the edge $(j_1, j_2) \in K$. Notice that $\mathsf{lift}(p) \in \mathsf{paths}(G, i_1, i_2)$.

We prove the correctness of Insert in five steps. First we present four lemmas and then the main result (Theorem 1). Each of the four lemmas states a key property of the Refine function. Before each lemma we will give an informal explanation that uses the following terminology. For a call $\mathsf{Refine}(G, A, O)$, we will refer to G as the *original* graph and we will refer to $\mathsf{Refine}(G, A, O)$ as the *refined* graph. Now let us move on to the four lemmas.

Intuitively, Lemma 8 says that reasoning about a path in the original graph carries over to the corresponding path in the refined graph.

Lemma 8. *Suppose* $G = (V, E, \ell)$ *and* $K \subseteq E$ *and* $(i_1, i_2) \in (V \times V)$ *and* $p \in$ *paths*$(\ \mathsf{Refine}(G, K), i_1, i_2)$. *If* $\mathsf{lift}(p), A \vdash i_1 \rightarrow i_2$, *then* $p, A \vdash i_1 \rightarrow i_2$.

Proof. We proceed by induction on the derivation of

$$\mathsf{lift}(p), A \vdash i_1 \rightarrow i_2.$$

We have two cases based on the last rule used in the derivation.

If the last rule used is the instantiation rule, then we have that i_1 can reach i_2 in $\mathsf{lift}(p)$, and we have $(L \mapsto R) \in A$, and $(L \mapsto R) \rhd (\ell(i_1), \ell(i_2))$. From $p \in \mathsf{paths}(\ \mathsf{Refine}(G, K), i_1, i_2)$, we have that i_1 can reach i_2 in p, so we can use the instantiation rule to derive $p, A \vdash i_1 \rightarrow i_2$.

If the last rule is the transitivity rule, then we can find j such that we can derive $\mathsf{lift}(p), A \vdash i_1 \longrightarrow j$ and $\mathsf{lift}(p), A \vdash j \longrightarrow i_2$. From the induction hypothesis, we have $p, A \vdash i_1 \longrightarrow j$ and $p, A \vdash j \longrightarrow i_2$. Now we use the transivity rule to derive $p, A \vdash i_1 \longrightarrow i_2$. □

Intuitively, Lemma 9 says that reasoning about the original graph carries over to the refined graph.

Lemma 9. *Suppose $G = (V, E, \ell)$ and $K \subseteq E$. If $G, A \models O$, then $\mathit{Refine}(G, K), A \models O$.*

Proof. Suppose $(i_1, i_2) \in O$, and let

$$p \in \mathsf{paths}(\ \mathsf{Refine}(G, K), i_1, i_2).$$

We have $\mathsf{lift}(p) \in \mathsf{paths}(G, i_1, i_2)$, so from $G, A \models O$, we have $\mathsf{lift}(p), A \vdash i_1 \longrightarrow i_2$, so from Lemma 8, we have $p, A \vdash i_1 \longrightarrow i_2$. □

Intuitively, Lemma 10 says that certain paths in the refined graph must contain a fence.

Lemma 10. $\forall (i_1, i_2) \in O$:
$\forall p \in \mathit{paths}(\ \mathit{Refine}(G,\ \mathit{Cut}(G, O)), i_1, i_2) :\ \exists j \in W_{\mathit{Cut}(G,O)} : j$ *is on* p.

Proof. Let $K = \mathsf{Cut}(G, O)$ and suppose also that $p \in \mathsf{paths}(\ \mathsf{Refine}(G, K), i_1, i_2)$. Notice $\mathsf{lift}(p) \in \mathsf{paths}(G, i_1, i_2)$. From the displayed Formula (2.2), we have $\mathsf{paths}((V, E \setminus K, \ell), i_1, i_2) = \emptyset$, so $\mathsf{lift}(p)$ contains at least one edge that is also an element of K. Let (j_1, j_2) be such an edge. The call $\mathsf{Refine}(G,\ \mathsf{Cut}(G, O))$ returns a graph that, among other things, adds a node v_{j_1,j_2} and replaces (j_1, j_2) with two edges. Notice that the node v_{j_1,j_2} is on p. Additionally, from the definition of W_K we have $v_{j_1,j_2} \in W_K$. So, we can choose $j = v_{j_1,j_2}$. □

Intuitively, Lemma 11 says that, with an appropriate assumption about the fence fany, the refined graph contains sufficient fences to enforce O.

Lemma 11. *If $\{(* \mapsto \mathit{fany}), (\mathit{fany} \mapsto *)\} \subseteq A$, then $\mathit{Refine}(G,\ \mathit{Cut}(G, O)), A \models O$.*

Proof. Suppose $(i_1, i_2) \in O$ and let furthermore $p \in \mathsf{paths}(\ \mathsf{Refine}(G,\ \mathsf{Cut}(G, O)), i_1, i_2)$. From Lemma 10, we have that we can find $j \in W_{\mathsf{Cut}(G,O)}$ such that j is on p. In particular, i_1 can reach j

in p, and j can reach i_2 in p, and $\ell(j) = \mathsf{fany}$. From $\ell(j) = \mathsf{fany}$ we have $(* \mapsto \mathsf{fany}) \triangleright (\ell(i_1), \ell(j))$ and $(\mathsf{fany} \mapsto *) \triangleright (\ell(j), \ell(i_2))$. From $\{(* \mapsto \mathsf{fany}), (\mathsf{fany} \mapsto *)\} \subseteq A$, we have that we can use the instantiation rule to get $p, A \vdash i_1 \longrightarrow j$ and $p, A \vdash j \longrightarrow i_2$. Now we use transitivity to conclude $p, A \vdash i_1 \longrightarrow i_2$. □

Now we are ready to prove the main result. Like Lemma 11, also Theorem 1 says that with an appropriate assumption about the fence fany, the refined graph contains sufficient fences to enforce O. The difference is that Lemma 11 is only about Cut and Refine, while Theorem 1 is about the entire definition of Insert, which also uses Elim.

Theorem 1. *If* $\{(* \mapsto \mathit{fany}), (\mathit{fany} \mapsto *)\} \subseteq A$*, then* $\mathit{Insert}(G, A, O), A \models O$.

Proof. Suppose $\{(* \mapsto \mathsf{fany}), (\mathsf{fany} \mapsto *)\} \subseteq A$ and let $G = (V, E, \ell)$. From the displayed Formula (2.1), we have $G, A \models \mathsf{Elim}(G, A, O)$, and from the displayed Formula (2.2), we have $\mathsf{Cut}(G, O \setminus \mathsf{Elim}(G, A, O)) \subseteq E$. When we apply Lemma 9 to those two properties, we get

$$\mathsf{Insert}(G, A, O), A \models \mathsf{Elim}(G, A, O). \tag{B.1}$$

From Lemma 11, we have:

$$\mathsf{Insert}(G, A, O), A \models O \setminus \mathsf{Elim}(G, A, O). \tag{B.2}$$

From the displayed Property (2.1), we have $\mathsf{Elim}(G, A, O) \subseteq O$, so:

$$\mathsf{Elim}(G, A, O) \cup (O \setminus \mathsf{Elim}(G, A, O)) = O. \tag{B.3}$$

From the displayed Formulas (B.1)–(B.3), we can conclude that $\mathsf{Insert}(G, A, O), A \models O$. □

APPENDIX C

Full Herd Model for the JAM

```
let opq = O | RA | V
let rel = W & RA
let acq = R & RA
let vol = V

(* release acquire ordering *)
let ra = po;[rel] | [acq];po

(* intra-thread volatile ordering *)
let volint =  po;[vol & R] | [vol & W];po

(* intra-thread ordering contraints *)
let into = svo | spush | ra | volint

(* define trace order, ensure it respects rf and intra-thread specified orders *)
(* Note that ((W * FW) & loc & ~id) = cofw *)
with to from linearisations(M\IW, ((W * FW) & loc & ~id) | rf | into)

(* cross thread push ordering extended with volatile memory accesses *)
let push = spush | volint
let pushto = to+ & (domain(push) * domain(push))

(* extend ra visibility *)
let vvo = rf | svo | ra | push | pushto;push
```

```
let vo = vvo+ | po-loc

include "filters.cat"
let WWco(rel) = WW(rel) & loc & ~id
let cofw = WWco((W * FW))

(* coherence rules *)
let coinit = loc & IW*(W\IW)
let coww = WWco(vo)
let cowr = WWco(vo;invrf)
let corw = WWco(vo;po)
let corr = WWco(rf;po;invrf)

(* general definition from RC11, works for atomic rws and split instruction rws *)
let rmw-jom = [RMW] | rmw

(* read-write rules *)
let cormwtotal = WWco(((range(rmw-jom) * _ ) | (_ * range(rmw-jom))) & to)
let cormwexcl = WWco((rf;rmw-jom)^-1;co-jom)

let rec co-jom = coww | cowr | corw  | corr
                | cofw | coinit | cormwtotal
                | WWco((rf;rmw-jom)^-1;co-jom)

acyclic (po | rf) & opq
acyclic co-jom
```

APPENDIX D

Specified Orders in the JAM: A Mapping for ARMv8 and x86 Read-writes

Specified orders can be compiled to existing synchronization using a fairly direct mapping. Here we give an example mapping for ARMv8. We also discuss how specified visibility orders can be elided when the target platform for compilation already enforces them.

$$[\mathtt{W}]; \mathsf{svo}; [\mathtt{M}] \rightsquigarrow [\mathtt{W}]; \mathsf{po}; [\mathtt{DMB\ ST}]; \mathsf{po}; [\mathtt{M}]$$

$$[\mathtt{R}]; \mathsf{svo}; [\mathtt{M}] \rightsquigarrow [\mathtt{R}]; \mathsf{po}; [\mathtt{DMB\ LD}]; \mathsf{po}; [\mathtt{M}]$$

$$[\mathtt{M}_1]; \mathsf{push}; [\mathtt{M}_2] \rightsquigarrow [\mathtt{M}_1]; \mathsf{po}; [\mathtt{DSB}]; \mathsf{po}; [\mathtt{M}_2]$$

The value of these orders can be seen in the case of atomic read-writes on x86. The JAM makes no intra-thread ordering guarantees for atomic read-writes even though they exist on some architectures like x86 [69]. This might result in unnecessary synchronization to achieve a desired outcome that requires such intra-thread ordering. However, using specified orders means that a compiler can make intelligent decisions based on the target platform. For example if we have a specified visibility order between a read-write and a later read target architecture is x86, the compiler can recognize that the head of the order is a compare and exchange instruction and omit any extra synchronization:

$$[\mathtt{RW}]; \mathsf{svo}; [\mathtt{R}] \rightsquigarrow [\mathtt{RW}]; \mathsf{po}; [\mathtt{R}]$$

APPENDIX E

Observable Total Coherence for ARMv8

Here we demonstrate that it is possible to construct a program that is only forbidden due to the total coherence order of the ARMv8 memory model from [74]. We start by noticing that Herd model for ARMv8 has two acyclicity requirements that involve co:

```
(* Internal visibility requirement *)
acyclic po-loc | ca | rf as internal

(* External visibility requirement *)
irreflexive ob as external
```

In the first requirement `ca = fr | co`. In the second `ob = obs | ...` where `obs = ... | coe` and coe is coherence restricted to inter-thread relationships. Critically, as illustrated in the proof for our model, they do not together work to form cycles. So we can use one with each side of the total order to demonstrate its observability in the model.

We have constructed the following program (included in the supplementary material) which will exhibit a cycle in the first requirement for one side of the total order and a different, incompatible cycle, for the second requirement:

```
Arch64 totalco
{
0:X1=x; 0:X3=y;
1:X1=x; 1:X3=y;
2:X1=x; 2:X3=y;
```

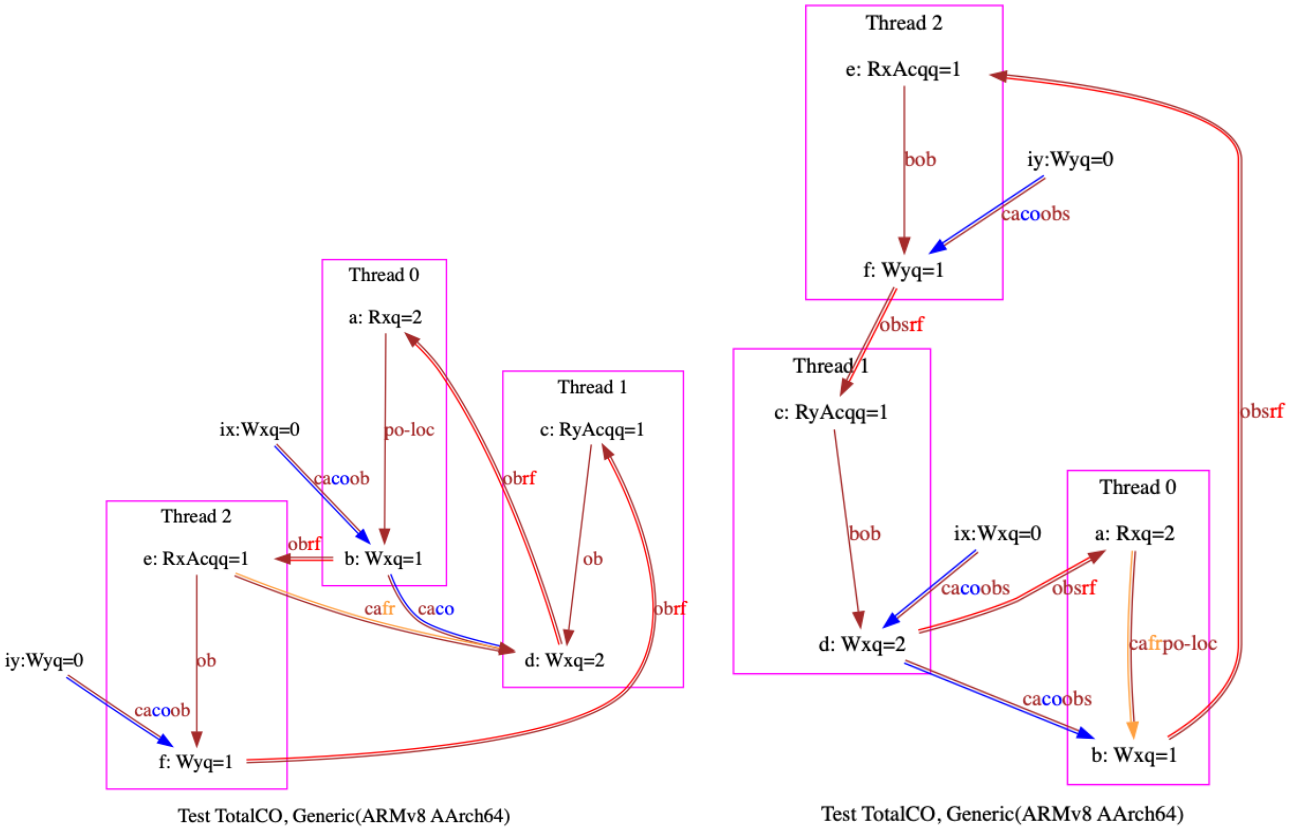
```
}
 P0            | P1           | P2;
 LDR X2,[X1]   | LDAR X5, [X3]| LDAR X5,[X1];
 MOV X0,#1     | MOV X2,#2    | MOV X0, #1;
 STR X0,[X1]   | STR X2,[X1]  | STR X0, [X3];

exists (0:X2=2 /\ 1:X5=1 /\ 2:X5=1)
```

We show this by running the program with Herd and allowing executions that violate these two requirements (we mark them with "flag" in the Herd parlance). The result in Figure E.1 is exactly two flagged executions. Each figure corresponds to one direction of the total ordering between b and d. We will inspect both to demonstrate that they are only forbidden as consequence of the total order, and thus if the total order was taken away they would both be allowed.

In Figure E.1a note the following. All of the cycles for any kind of edge involve a $\xrightarrow{\texttt{po-loc}}$ b which is not in ob. This means we can avoid a cycle in ob. Recall that if there were a cycle in ob then when we use ob for the other side of the total coherence order it would be a cycle regardless of the total coherence order. Also, note that if we take away b $\xrightarrow{\texttt{co}}$ d (blue) it will also remove e $\xrightarrow{\texttt{fr}}$ d. Thus, if we take away b $\xrightarrow{\texttt{co}}$ d by removing the total coherence order of the model, there is no cycle left in the graph and the execution would be allowed.

In Figure E.1b note the following. All of the cycles for any kind of edge involve c $\xrightarrow{\texttt{bob}}$ d which is not in `po-loc | ca | rf`. This means we can't establish a cycle in `po-loc | ca | rf`. Thus, if we take away d $\xrightarrow{\texttt{co}}$ b by removing the total coherence order of the model, there is no cycle in ob left in the graph and the execution would be allowed.



(a) `po-loc | ca | rf` Cycle

(b) `ob` Cycle

Figure E.1: Two Flagged Executions

APPENDIX F

Full Expression Semantics

The following constitutes the full expression semantics of our model of the JOM.

F.1 Expression Rules

$$\frac{k = n + m}{n + m \xrightarrow{\emptyset} k}\ \text{add} \qquad \frac{k = n \bmod m}{n \bmod m \xrightarrow{\emptyset} k}\ \text{mod} \qquad \frac{n = m}{n == m \xrightarrow{\emptyset} 1}\ \text{eq} \qquad \frac{n \neq m}{n == m \xrightarrow{\emptyset} 0}\ \text{neq}$$

$$\frac{e_1 \xrightarrow{d} e_1'}{\texttt{let } x := e_1 \texttt{ in } e_2 \xrightarrow{d} \texttt{let } x := e_1' \texttt{ in } e_2}\ \text{let-left} \qquad \frac{e_2 \xrightarrow{d} e_2' \quad e_1 \texttt{ init} \quad x \notin FV(d)}{\texttt{let } x := e_1 \texttt{ in } e_2 \xrightarrow{d} \texttt{let } x := e_1 \texttt{ in } e_2'}\ \text{let-right}$$

$$\frac{}{\texttt{let } x := n \texttt{ in } e_2 \xrightarrow{\emptyset} [n/x]e_2}\ \text{let-subst} \qquad \frac{}{\texttt{repeat } e \texttt{ end} \xrightarrow{\emptyset} \texttt{let } x := e \texttt{ in if } x \texttt{ then } x \texttt{ else repeat } e \texttt{ end}}\ \text{repeat}$$

$$\frac{n \neq 0}{\texttt{if } n \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\emptyset} e_1}\ \text{if-right} \qquad \frac{}{\texttt{if } 0 \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\emptyset} e_2}\ \text{if-left}$$

$$\frac{}{l{:}n \xrightarrow{\emptyset} n}\ \text{label-rm} \qquad \frac{e \xrightarrow{d} e'}{l{:}e \xrightarrow{d'} l{:}e'}\ \text{label-under}$$

$$\frac{}{[s/x]e \xrightarrow{\emptyset} s = n \texttt{ in } [n/x]e}\ \text{spec} \qquad \frac{e \xrightarrow{d} e'}{s = n \texttt{ in } e \xrightarrow{d} s = n \texttt{ in } e'}\ \text{spec-under} \qquad \frac{}{n = n \texttt{ in } e \xrightarrow{\emptyset} e}\ \text{spec-rm}$$

$$\frac{}{a \xrightarrow{\epsilon:i=a} i}\ \text{action-init} \qquad \frac{}{i \xrightarrow{i \text{ to } n} n}\ \text{exec-read} \qquad \frac{}{i \xrightarrow{i} 0}\ \text{exec-write}$$

F.2 State and Thread Rules

$$\frac{P \xrightarrow{d@p} P' \quad H \xrightarrow{d@p} H'}{(P, H) \to (P', H')}\ \text{step}$$

$$\frac{e \xrightarrow{d} e'}{P; p : \texttt{fork}\ e \xrightarrow{d@p} P; p : \texttt{fork}\ e'}\ \text{P-step} \qquad \frac{P \xrightarrow{d@p_1} P'}{P; p_2 : \texttt{fork}\ e \xrightarrow{d@p_1} P; p_2 : \texttt{fork}\ e}\ \text{P-find}$$

F.3 Init Rules

The intialization rules and the use of `init` in let-right rule forces initialization occur in program order.

$$\frac{}{n\ \text{init}}\ \text{init-nat} \qquad \frac{e_1\ \text{init} \quad e_2\ \text{init}}{\texttt{let}\ x := e_1\ \texttt{in}\ e_2\ \text{init}}\ \text{init-let} \qquad \frac{}{i\ \text{init}}\ \text{init-id}$$

$$\frac{e\ \text{init}}{s = n\ \texttt{in}\ e\ \text{init}}\ \text{init-spec} \qquad \frac{e\ \text{init}}{s = n\ \texttt{in}\ e\ \text{init}}\ \text{init-spec} \qquad \frac{e\ \text{init}}{l : e\ \text{init}}\ \text{init-label}$$

APPENDIX G

Full Logic

The following consitutes all the deduction rules in our logic.

G.1 Core

$$\Gamma \vdash a_1 \vee a_2 \qquad \Gamma \vdash a_1 \Rightarrow a_3 \qquad \frac{\Gamma \vdash a_2 \Rightarrow a_3}{\Gamma \vdash a_3}\text{disj-elim} \qquad \frac{\Gamma \vdash a_1}{\Gamma \vdash a_1 \vee a_2}\text{disj-lintro} \qquad \frac{\Gamma \vdash a_2}{\Gamma \vdash a_1 \vee a_2}\text{disj-rintro}$$

$$\Gamma \vdash a_1 \vee a_2 \qquad \Gamma \vdash a_1 \Rightarrow a_2$$

$$\frac{\Gamma \vdash \neg a_1}{\Gamma \vdash a_2}\text{disj-syl} \qquad \frac{\Gamma \vdash \text{false}}{\Gamma \vdash a}\text{exfalso} \qquad \frac{(\Gamma; a_1 \vdash a_2)}{\Gamma \vdash a_1 \Rightarrow a_2}\text{impl-intro} \qquad \frac{\Gamma \vdash a_1}{\Gamma \vdash a_2}\text{mp}$$

$$\Gamma \vdash a_2 \qquad \frac{\Gamma \vdash a_1}{\Gamma \vdash a_1 \wedge a_2}\text{conj-intro} \qquad \frac{\Gamma \vdash a_1 \wedge a_2}{(\Gamma \vdash a_1)}\text{conj-lelim} \qquad \frac{\Gamma \vdash a_1 \wedge a_2}{\Gamma \vdash a_2}\text{conj-relim}$$

G.2 Inequality

$$\frac{\Gamma \vdash V_1 \doteq V_2}{\Gamma \vdash V_2 \doteq V_1}\text{geq-sym} \qquad \frac{\Gamma \vdash V_1 \doteq V_2 \qquad \Gamma \vdash V_2 \doteq V_3}{\Gamma \vdash V_1 \doteq V_3}\text{geq-trans} \qquad \frac{\Gamma \vdash L \doteq n_1 \qquad \Gamma \vdash L \doteq n_2 \qquad n_1 \neq n_2}{\Gamma \vdash \mathsf{false}}\text{nope}$$

$$\frac{}{\Gamma \vdash \neg V \dot{<} V}\text{glt-irr} \qquad \frac{\Gamma \vdash V_1 \dot{<} V_3 \qquad \Gamma \vdash V_3 \dot{<} V_2}{\Gamma \vdash V_1 \dot{<} V_2}\text{glt-trans} \qquad \frac{\Gamma \vdash V_1 \dot{<} V_2}{\Gamma \vdash \neg V_2 \dot{<} V_1}\text{glt-asym}$$

G.3 Expressions

$$\frac{\Gamma \vdash n \mathbin{@} L}{\Gamma \vdash L \doteq n}\text{const} \qquad \frac{\Gamma \vdash (L; l_{opr}) \doteq n_2}{\Gamma \vdash L \doteq (n_1[+]n_2)}\text{add} \qquad \frac{\Gamma \vdash n_1 = n_2}{\Gamma \vdash L \doteq 1}\text{eq}$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 \,\mathsf{mod}\, s_2 \mathbin{@} L \\ \Gamma \vdash (L; l_{opl}) \doteq n_1 \\ \Gamma \vdash (L; l_{opr}) \doteq n_2\end{array}}{\Gamma \vdash L \doteq (n_1\ [\mathsf{mod}]\ n_2)}\text{mod} \qquad \frac{\Gamma \vdash (\texttt{if}\ x\ \texttt{then}\ l_1 : e_1\ \texttt{else}\ l_2 : e_2 \mathbin{@} L}{\Gamma \vdash ((\neg(L; l_{cnd}) \doteq 0) \wedge (L; l_1) \doteq L) \vee (L; l_{cnd}) \doteq 0 \wedge (L; l_2) \doteq L}\text{if}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{let}\ x := l_1 : e_1\ \texttt{in}\ l_2 : e_2 \mathbin{@} L \\ \Gamma \vdash x \mathbin{@} (L; l_2; ; L')\end{array}}{\Gamma \vdash (L; l_2; ; L') \doteq (L; l_1)}\text{let-bind} \qquad \frac{\Gamma \vdash \texttt{let}\ x := l_1 : e_1\ \texttt{in}\ l_2 : e_2 \mathbin{@} L}{\Gamma \vdash \exists n, (L; l_1) \doteq n}\text{let-left}$$

$$\frac{\Gamma \vdash \texttt{let}\ x := l_1 : e_1\ \texttt{in}\ l_2 : e_2 \mathbin{@} L}{\Gamma \vdash (L; l_2) \doteq L}\text{let-right} \qquad \frac{\Gamma \vdash \texttt{repeat}\ e\ \texttt{end} \mathbin{@} L}{\Gamma \vdash \exists l, e \mathbin{@} (L; l)}\text{repeat-match}$$

$$\frac{\Gamma \vdash \texttt{repeat}\ e\ \texttt{end} \mathbin{@} L}{\Gamma \vdash \exists l, L \doteq (L; l) \wedge (\neg(L; l) \doteq 0)}\text{repeat-break} \qquad \frac{\Gamma \vdash (\exists n, L \doteq n)}{\Gamma \vdash \exists e, e \mathbin{@} L}\text{geq-match}$$

$$\frac{\Gamma \vdash [s_1] \mathbin{@} L \qquad \Gamma \vdash [s_1] := s_2 \mathbin{@} L}{\Gamma \vdash (L; l_{loc}) \doteq l \qquad \Gamma \vdash (L; l_{loc}) \doteq l}$$

$$\frac{\Gamma \vdash L \doteq n}{\Gamma \vdash \mathsf{reads}(L, l, n)}\text{reads} \qquad \frac{\Gamma \vdash (L; l_{val}) \doteq n}{\Gamma \vdash \mathsf{writes}(L, l, n)}\text{writes}$$

G.4 Memory

$$\frac{\Gamma \vdash \texttt{let}\ x := l_1 : e_1\ \texttt{in}\ l_2 : e_2\ @\ L}{\Gamma \vdash (L; l_1) \xrightarrow{\texttt{po}} (L; l_2)}\text{po} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\texttt{po}} L_2}{\Gamma \vdash (L_1;;L_3) \xrightarrow{\texttt{po}} (L_2;;L_4)}\text{po-sub}$$

$$\Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_3$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\texttt{rf}} L_2}{\Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_2}\text{vo-rf} \qquad \frac{\Gamma \vdash L_3 \xrightarrow{\texttt{vo}} L_2}{\Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_2}\text{vo-trans}$$

$$\Gamma \vdash L_1 \xrightarrow{\texttt{push}} L_2$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\texttt{po}} L_2}{\Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_2}\text{vo-po} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_2}{\Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_2}\text{vo-vo} \qquad \frac{\Gamma \vdash L_3 \xrightarrow{\texttt{push}} L_4}{\Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_4 \vee L_3 \xrightarrow{\texttt{vo}} L_2}\text{vo-pushes}$$

$$\Gamma \vdash \text{writes}(L_1, l, n_1) \qquad \Gamma \vdash L_1 \neq L_2$$

$$\Gamma \vdash \text{reads}(L_r, l, n_2) \qquad \Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_r \quad \Gamma \vdash \text{writes}(L_1, l, n_1)$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_2 \quad \Gamma \vdash \text{writes}(L_2, l, n_3)}{\Gamma \vdash L_1 \xrightarrow{\texttt{co}} L_2}\text{co-ww} \qquad \frac{\Gamma \vdash L_r \xrightarrow{\texttt{po}} L_2 \quad \Gamma \vdash \text{reads}(L_r, l, n_2)}{\Gamma \vdash L_1 \xrightarrow{\texttt{co}} L_2}\text{co-wr}$$

$$\Gamma \vdash L_1 \xrightarrow{\texttt{rf}} L_{r_1} \qquad \Gamma \vdash L_1 \neq L_2$$

$$\Gamma \vdash L_1 \xrightarrow{\texttt{vo}} L_r \quad \Gamma \vdash \text{writes}(L_1, l, n_1) \qquad \Gamma \vdash L_2 \xrightarrow{\texttt{rf}} L_{r_2} \quad \Gamma \vdash \text{writes}(L_1, l, n_1)$$

$$\frac{\Gamma \vdash L_r \xrightarrow{\texttt{po}} L_2 \quad \Gamma \vdash \text{writes}(L_2, l, n_2)}{\Gamma \vdash L_1 \xrightarrow{\texttt{co}} L_2}\text{co-rw} \qquad \frac{\Gamma \vdash L_{r_1} \xrightarrow{\texttt{po}} L_{r_2} \quad \Gamma \vdash \text{writes}(L_2, l, n_2)}{\Gamma \vdash L_1 \xrightarrow{\texttt{co}} L_2}\text{co-rr}$$

$$\frac{\Gamma \vdash \text{reads}(L_r, l, n)}{\Gamma \vdash \exists L_w, L_w \xrightarrow{\texttt{rf}} L_r}\text{reads-rf} \qquad \frac{\Gamma \vdash L_w \xrightarrow{\texttt{rf}} L_r}{\Gamma \vdash \exists\ l\ n, \text{writes}(L_w, l, n) \wedge \text{reads}(L_r, l, n)}\text{rf}$$

$$\Gamma \vdash L_1 \xrightarrow{\texttt{co}} L_2$$

$$\frac{\Gamma \vdash L_2 \xrightarrow{\texttt{co}} L_1}{\Gamma \vdash \text{false}}\text{co-cycl} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\texttt{coi}} L_2}{\Gamma \vdash \neg(L_1 \xrightarrow{\texttt{co}} L_3 \wedge L_3 \xrightarrow{\texttt{co}} L_2)}\text{coi} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\texttt{coi}} L_2}{\Gamma \vdash L_1 \xrightarrow{\texttt{co}} L_2}\text{coi-co}$$

$$\Gamma \vdash L_1 \xrightarrow{\texttt{co}} L_2$$

$$\Gamma \vdash (L_1 \xrightarrow{\texttt{coi}} L_2) \Rightarrow P\ L_1\ L_2$$

$$\frac{(\forall L_3, \Gamma \vdash L_1 \xrightarrow{\texttt{co}} L_3 \Rightarrow P\ L_1\ L_3 \Rightarrow L_3 \xrightarrow{\texttt{coi}} L_2 \Rightarrow P L_1 L_2)}{\Gamma \vdash P\ L_1\ L_2}\text{co-ind}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\texttt{co}} L_2}{\Gamma \vdash \exists l\, n_1\, n_2,\ \mathsf{writes}(L_1, l, n_1) \wedge \mathsf{writes}(L_2, l, n_2)}\text{co-writes}$$

$$\frac{\Gamma \vdash \mathsf{reads}(L, l_1, n_1) \qquad \Gamma \vdash \mathsf{reads}(L, l_2, n_2)}{\Gamma \vdash l_1 = l_2}\text{reads-match-loc}$$

$$\frac{\Gamma \vdash \mathsf{reads}(L, l_1, n_1) \qquad \Gamma \vdash \mathsf{reads}(L, l_2, n_2)}{\Gamma \vdash n_1 = n_2}\text{reads-match-val}$$

$$\frac{\Gamma \vdash \mathsf{writes}(L, l_1, n_1) \qquad \Gamma \vdash \mathsf{writes}(L, l_2, n_2)}{\Gamma \vdash l_1 = l_2}\text{writes-match-loc}$$

$$\frac{\Gamma \vdash \mathsf{writes}(L, l_1, n_1) \qquad \Gamma \vdash \mathsf{writes}(L, l_2, n_2)}{\Gamma \vdash n_1 = n_2}\text{writes-match-val}$$

APPENDIX H

The match Function and actionid Predicate

Note, for the full match function consult the `exmatch` definition in the `Semantics.v` Coq source file in the supplementary material. Here we have elided the parts of the definition that traverses the head of the executions and threads.

$$\mathsf{match}(e, L, e') \triangleq \begin{cases} e = e' & \text{if } L = \emptyset \\ \mathsf{match}(s_1, L', e') & \text{if } e = s_1 + s_2 \wedge L = l_{lopl}; L' \\ \mathsf{match}(s_1, L', e') & \text{if } e = s_1 \ \bmod\ s_2 \wedge L = l_{lopl}; L' \\ \mathsf{match}(s_1, L', e') & \text{if } e = s_1 == s_2 \wedge L = l_{lopl}; L' \\ \mathsf{match}(s_2, L', e') & \text{if } e = s_1 + s_2 \wedge L = l_{lopr}; L' \\ \mathsf{match}(s_2, L', e') & \text{if } e = s_1 \ \bmod\ s_2 \wedge L = l_{lopr}; L' \\ \mathsf{match}(s_2, L', e') & \text{if } e = s_1 == s_2 \wedge L = l_{lopr}; L' \\ \mathsf{match}(s, L', e') & \text{if } e = \texttt{[}s\texttt{]} \wedge L = l_{lloc}; L' \\ \mathsf{match}(s_1, L', e') & \text{if } e = \texttt{[}s_1\texttt{]} := s_2 \wedge L = l_{lloc}; L' \\ \mathsf{match}(s_2, L', e') & \text{if } e = \texttt{[}s_1\texttt{]} := s_2 \wedge L = l_{lval}; L' \\ \mathsf{match}(e'', L', e') & \text{if } e = l{:}e'' \wedge L = l; L' \\ \mathsf{match}(e'', L, e') & \text{if } e = \texttt{repeat } e'' \texttt{ end} \\ \mathsf{match}(s, L', e') & \text{if } e = \texttt{if } s \texttt{ then } e_1 \texttt{ else } e_2 \wedge L = l_{lcnd}; L' \\ \mathsf{match}(e_1, L, e') & \text{if } e = \texttt{if } s \texttt{ then } e_1 \texttt{ else } e_2 \wedge \neg\mathsf{match}(e_2, L, e') \\ \mathsf{match}(e_2, L, e') & \text{if } e = \texttt{if } s \texttt{ then } e_1 \texttt{ else } e_2 \\ \mathsf{match}(e_1, L, e') & \text{if } e = \texttt{let } x := e_1 \texttt{ in } e_2 \wedge \neg\mathsf{match}(e_2, L, e') \\ \mathsf{match}(e_2, L, e') & \text{if } e = \texttt{let } x := e_1 \texttt{ in } e_2 \\ \mathsf{false} & \text{o/w} \end{cases}$$

As discussed in Section 4.2, the actionid predicate uses match to track the evolution of a memory access through an execution from expression to identifier to value.

$$
\begin{aligned}
&\mathsf{actionid}(E, L, i) \triangleq \\
&\quad \exists\, e\, n\, E'\, E'', \\
&\qquad E = E_1;;E_2;;E_3 \\
&\qquad e \in \{[s], [s_1] := s_2\} \\
&\qquad \mathsf{match}(E, L, e) \\
&\qquad \mathsf{match}(E_2, L, i) \\
&\qquad \mathsf{match}(E_3, L, n)
\end{aligned}
$$

APPENDIX I

Soundness Examples

We will focus here on proof sketches for the soundness of a few key rules we have featured previously. We refer the interested reader to the supplementary material for more extensive, formal proofs.

reads-rf By assumption we have that a read, L_r evaluated completely. We must show that it is connected to a write that can be located in the program with some L_w. We know that if a read evaluated to a value then it must have some identifier i_r and by hist-read in Figure 3.19 it will have added $\mathsf{rf}(i_w, i_r)$ to the history. Thus by the assumptions of hist-read, there was some write identified by i_w that executed fully. By the assumption of our labeling discipline we have that it must have been located in the program at the point of it executed with some L_w. All that remains is to construct $L_w \xrightarrow{\mathtt{rf}} L_r$ using the write and read labels with the derived information about their executions for the actionid predicate.

co-ind By assumption $L_1 \xrightarrow{\mathtt{co}} L_2$. By the definition of $\xrightarrow{R}$ we have that there exists some i_1 and i_2 for L_1 and L_2 such that $i_1 \xrightarrow{\mathtt{co}}_E i_2$. We know that $\xrightarrow{\mathtt{co}}_E$ is well founded since it is acyclic (see Figure 3.19 hist-write and hist-read) and E is finite. The proof proceeds by induction on $i_1 \xrightarrow{\mathtt{co}}_E i_2$ to show $P'\ i_1\ i_2 \triangleq P\ L_1\ L_2$. Note that, since neither i_1 nor i_2 can be free in P by the syntax of our assertions we can simply use $P\ L_1\ L_2$.

In the base case show, $i_1 \xrightarrow{\mathtt{coi}}_E i_2 \Rightarrow P\ L_1\ L_2$. By assumption, $L_1 \xrightarrow{\mathtt{coi}} L_2 \Rightarrow P\ L_1\ L_2$, so it's enough to show that $i_1 \xrightarrow{\mathtt{coi}}_E i_2 \Rightarrow L_1 \xrightarrow{\mathtt{coi}} L_2$, which follows from the definition of $\xrightarrow{R}$ and the assumptions from the definition of $L_1 \xrightarrow{\mathtt{co}} L_2$. In the inductive case we have as an assumption $P'\ i_1\ i_3$ for i_1 and some i_3. According to the the syntax of our assertions we have immediately that $P\ L_1\ L_2$.

if-alt By assumption we have $L \doteq n$ for some n and $\mathtt{if} \ldots @ L$. Then, by the definition of E

we have some, possibly empty, sequence of steps, $(P_1, H_1) \rightarrow (P_2, H_2)$ from the head of E where `if` ... $@\ L$ to the state where $n\ @\ L$. We proceed by induction on execution steps to show that, if there is a match `if` ... $@\ L$ in the first state (holding the subexpressions to be arbitrary) and a match $n\ @\ L$ in the end state, then there exist some pair of states where the disjunction in the conclusion of if-alt holds.

In the base case E is only the initial state. Then both `if` ... $@\ L$ and $n\ @\ L$ would be true in the same state implying that `if` ... $= n$, a contradiction.

In the inductive case, we have $(P_1, H_1) \rightarrow (P_3, H_3) \rightarrow^* (P_2, H_2)$. We know that `if` ... $@\ L$ in P_1 then it must be that there is some $e\ @\ L$ in P_3. Since the equality of expressions is decideable we consider the cases. If `if` ... $= e$ then the inductive hypothesis applies. Otherwise it must be that there was a substitution or the `if` expression, took a step. Again, we consider the cases. In the case of the substitution we can apply the inductive hypothesis since we held the sub-expressions to be arbitrary in our goal. Otherwise, the `if` took a step. Then we have the two states, (P_1, H_1) and (P_3, H_3), where either, the condition was `1` in P_1 and the `then` branch will be the resulting expression at label L in P_3, or, similarly, the condition was `0` and the `else` branch will be the resulting expression at label L, as required.

APPENDIX J

Herlihy/Wing Queue Correctness

The concurrent queue specification of Herlihy and Wing [36] tracks closely with the ordering based approach of memory reasoning in our logic. Here we give short proof sketches for how Lemma 5 can be used to prove the key theorems of their specification.

Theorem 6 Since the `Enq x` "precedes" the `Enq y` (here $\xrightarrow{\texttt{vo}}$) we can show that the index for the first is smaller than the index for the second by Lemma 6. By Lemma 5 the index of `Deq x` must be smaller than the index `Deq y` therefore we can show that `Deq x` must also have happened before `Deq y` using `tryCons` invariant.

Theorem 7 This is similar to Theorem 9 in that we must show that there is a dequeue (`tryCons`) which executed with the same index as the earlier dequeue. The proof here is easier. The proof of our Theorem 9 must establish that there exists a dequeue action which increments the reader index, allowing the write of writer index modulo N in the second dequeue to reference the same buffer location. Instead, we are given the second dequeue, so we only need to "work backward" using the `tryCons` invariant to show that there must exist an earlier dequeue with the same index as the earlier enqueue and use Lemma 5 to show that it must have read from the earlier enqueue.

Theorem 8 Follows directly from the $\xrightarrow{\texttt{rf}}$ established during an enqueue for a given buffer location and the definition of $\xrightarrow{\texttt{vo}}$.