Verification of Concurrent Data Structures with TLA

Zixian Cai

A report submitted for the course
SCNC2102 Advanced Studies 2
The Australian National University

November 2018

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this report is my own original work.

Zixian Cai
9 November 2018

Acknowledgments

First, I would like to thank my supervisor, Michael Norrish. We walked through *Logical Foundations*[1] last summer, where I got to know the formal side of software construction. Logical and "illogical" discussions with you are always enjoyable. This project would not have been possible with your patient guidance and inspiration.

I would also like to thank the TLA+ community, particularly Leslie Lamport and Stephan Merz, for their help on using the TLA+ Model Checker (TLC).

Thank you for being there during difficult times, Brenda Wang. For the feedback you gave on draft versions of this report I am also deeply grateful. Hope you get as much happiness as you gave me and best luck with your own project.

[1] https://softwarefoundations.cis.upenn.edu

Abstract

Concurrent systems have critical applications such as aviation. Due to their inherent complexity, mechanised verification is well suited for reasoning about the safety and liveness properties of such systems. Temporal logics, such as TLA and LTL, have been used to verify distributed systems and protocols. However, it is not clear whether these logics are good fits for modelling and verifying concurrent data structures.

This work describes how Temporal Logic of Actions (TLA) can be adapted to handle concurrent data structures and weak memory models. It also shows how the TLA toolchain, especially the model checker, can aid in cleanly applying the logical machinery to concrete programs.

I used litmus tests to validate my encoding of memory models against prior work. These models enabled me to formalize various concurrent data structures, including the Chase-Lev queue. Then, I am able to check the behaviours of these data structures against abstract specification of their operations. In particular, my modelling can successfully find bugs in a faulty implementation of the Chase-Lev queue.

The results suggest that TLA is appropriate for modelling concurrent data structures. The formal models I designed and the related modelling techniques can be used by the wider research community in their verification work.

Contents

List of Figures

Chapter 1

Introduction

Concurrent programs are notoriously hard to write, where hard-to-reproduce and subtle bugs often manifest. This report focuses on specifying and verifying concurrent data structures using Temporal Logic of Actions (TLA).

1.1 Problem Statement

Real-world systems are inherently concurrent, and computer systems are no exception. With the prevalent use of concurrent hardware nowadays, we need concurrent programs to utilize the available concurrency. And proper synchronization is often necessary for these programs to produce correct results.

Coarse-grained synchronization is a naïve way to ensure correctness. For example, we can make a data structure thread-safe by using a mutex around the entire object. However, this approach hinders scalability due to unnecessary synchronization.

Fine-grained synchronization mechanisms, such as the ones based on the compare-and-swap operation, make programs more scalable. However, these mechanisms are more difficult to reason about. Instead of dealing with a single, big critical section, we need to consider more possible interleaving of operations. Even worse, the outcomes of interleaving are often unexpected due to weak memory models that exhibits on many hardware architectures.

Traditional approaches to reasoning about programs, such as pen-and-paper proofs, do no scale up to the complexity we see in concurrent programs. Thus, we need more scalable tools to help us understand the behaviours of large, real-world concurrent programs. Several logic systems have been proposed to help such reasoning, including concurrent separation logic [O'Hearn, 2007], linear temporal logic [Pnueli, 1977], and rely-guarantee [Jones, 1983].

The Temporal Logic of Actions [Lamport, 1994] is widely used in the field of distributed systems and protocols. However, it is not clear whether TLA is well suited to specify and verify concurrent data structures.

1.2 Contribution

I selected and modelled several classic concurrent data structures using TLA constructs. I also investigated weak memory models and encoded TSO and WMM in TLA. The correctness of these encodings are cross-checked against prior work through litmus tests. After integrating the above modelling, I am then able to reason about the behaviours of different concurrent data structures on different memory models. In particular, I successfully found a bug in an implementation of the Chase-Lev queue.

1.3 Outline

Chapter 2 gives an overview of TLA, which is the logic system used in this work. Chapter 3 introduces techniques to encode different components of concurrent data structures in TLA. Chapter 4 discusses memory models and how weak memory models can be implemented in TLA. Then, Chapter 5 uses a concrete example, the Chase-Lev queue, to demonstrate how the above modelling techniques can be applied to complex data structures. Finally, Chapter 6 summaries the contribution of this work and points out future work.

Chapter 2

Temporal Logic of Actions

Prose, which is prone to subtle errors, is a poor medium for conducting formal, rigorous reasoning. Proposed by Lamport, the Temporal Logic of Actions (TLA) [Lamport, 1994] is a simple yet flexible logic system to specify concurrent systems. Contrasting with Linear Temporal Logic (LTL) [Pnueli, 1977], TLA avoids the use of temporal operators where possible. This is to reduce the complication involving in temporal reasoning compared with nonmodal, first-order reasoning. In this chapter, I will summarize important concepts of TLA, enabling the reader to follow the development of modelling in Chapter 3 and Chapter 4.

2.1 States and Actions

A program can be described by the states it maintains, and the series of actions it performs to manipulate these states. Therefore, it is important that we express these two aspects in TLA.

The *state* of a program gives meaning to variables by mapping them to values.

$$\mathbf{St} :: \mathbf{Var} \to \mathbf{Val} \tag{2.1}$$

The meaning of a variable x in a state s is denoted by $s[\![x]\!]$. Then, the meaning of expressions, which consist of variables and constants, can be defined in the following way. For an expression f, its value in a state s can be obtained by replacing each variable v by its evaluation in that state.

$$s[\![f]\!] \triangleq f(\forall v : s[\![v]\!]/v) \tag{2.2}$$

An *action* changes the state of a program. We can represent an action by a relation between old states and new states using a boolean expression. This boolean expression can consist of variables, primed variables and constants, where primed variables concern the new states and unprimed variables concern the old states. The semantics $[\![\mathcal{A}]\!]$ of an action $\mathcal{A}$ is defined by evaluating each of the unprimed variables and the primed variables in the old state and the new state respectively.

$$s[\![\mathcal{A}]\!]t \triangleq \mathcal{A}(\forall v : s[\![v]\!]/v, t[\![v]\!]/v') \tag{2.3}$$

For example, for action $x' + x < 1$, $s[\![x' + x < 1]\!]t$ evaluates to $t[\![x]\!] + s[\![x]\!] < 1$. We call a pair of state (s, t) an $\mathcal{A}$ step if and only if $s[\![\mathcal{A}]\!]t$.

Actions map naturally to operations of programs. That is, if $\mathcal{A}$ corresponds to an operation of a program, then its execution in state s can lead to state t if (s, t) is an $\mathcal{A}$ step. For example, incrementing a variable x can be represented by $\mathcal{A} \triangleq x' = x + 1$, where $([x \to 41], [x \to 42])$ is an $\mathcal{A}$ step.

2.2 Temporal Formulae

When reasoning about behaviours of programs, one often needs to consider executions, which are sequences of states.

$[\![\mathcal{A}]\!]$ is true for a behaviour if and only if the first pair of states is an $\mathcal{A}$ step. That is, for a behaviour σ, which represents sequence $< s_0, s_1, \ldots >$, we have the following

$$\sigma[\![\mathcal{A}]\!] =< s_0, s_1, \ldots > [\![\mathcal{A}]\!] \triangleq s_0[\![\mathcal{A}]\!]s_1 \tag{2.4}$$

Compound formulae are defined in terms of elementary formulae.

$$\begin{aligned} \sigma[\![F \wedge G]\!] &\triangleq \sigma[\![F]\!] \wedge \sigma[\![G]\!] \\ \sigma[\![\neg F]\!] &\triangleq \neg\sigma[\![F]\!] \end{aligned} \tag{2.5}$$

An F is *always* ($\Box$) true if it is true at every single point of "time".

$$< s_0, s_1, \ldots > [\![\Box F]\!] \triangleq \forall n \in \text{Nat} :< s_n, s_{n+1}, \ldots > [\![F]\!] \tag{2.6}$$

An F that *eventually* ($\Diamond$) holds if it is not that case that F is always false ($\neg\Box\neg F$). That is, there will be a point of "time" such that F holds.

$$< s_0, s_1, \ldots > [\![\Diamond F]\!] \equiv \exists n \in \text{Nat} :< s_n, s_{n+1}, \ldots > [\![F]\!] \tag{2.7}$$

It is sometimes desirable to make assumptions and exclude some unreasonable executions. For example, if we have two or more processes, a reasonable scheduler will allocate execution times to each process appropriately, and the executions should be excluded where one process makes no progress even when it is possible to do so. We call these assumptions fairness assumptions.

Before defining fairness, let's define what it means that a program can make some progress. An action $\mathcal{A}$ is enabled in a state if and only if it is possible to make an $\mathcal{A}$ step in that state.

$$s[\![\text{Enabled } \mathcal{A}]\!] \triangleq \exists t \in \mathbf{St} : s[\![\mathcal{A}]\!]t \tag{2.8}$$

An execution is strongly $\mathcal{A}$-fair if $\mathcal{A}$ must be taken when it is infinitely often possible to do so.

$$\text{SF}(\mathcal{A}) \triangleq (\Box\Diamond\text{Enabled } \mathcal{A}) \implies (\Box\Diamond\mathcal{A}) \tag{2.9}$$

An execution is weakly $\mathcal{A}$-fair if $\mathcal{A}$ must be taken when it is always possible to do so.

$$\mathrm{WF}(\mathcal{A}) \triangleq (\Diamond\Box\text{Enabled } \mathcal{A}) \implies (\Box\Diamond\mathcal{A}) \tag{2.10}$$

Note that strong fairness implies weak fairness.

Fairness is especially important if we try to verify liveness properties. For example, for mutual exclusion, one might want to specify that each waiting process should eventually enter its critical section. However, if the action of entering critical section is not strongly fair, a process might stay in waiting states due to poor scheduling despite being infinitely often possible to enter the critical section.

2.3 Model Checking

The specification language of TLA is called TLA+ [Lamport, 2002]. I will not discuss its formal grammar and constructions here, and curious readers are encouraged to read relevant chapters in [Lamport, 2002]. In practise, it is often more convenient and concise to express systems in PlusCal [Lamport, 2009], which can be translated to TLA+ for later use. PlusCal is heavily used in this work to express concurrent data structures and operational models of memory models.

For systems and properties written in TLA+, there are two widely used tools to help us verify the correctness of systems. The TLA+ Proof System (TLAPS) [Chaudhuri et al., 2010] can be used to conduct machine-checked proofs. Another is the TLA+ Model Checker (TLC) [Lamport, 2002], which is used in this work.

The model checking process aims to answer whether a system conforms to the given specifications, and produces counterexamples if it is not the case. TLC employs directed graphs as the data structure for maintaining the reachability of states. Each node of the graph corresponds to a state, and each edge from state s to state t represents that (s, t) is a valid step for some actions.

To explore reachable states, classic graph traversal algorithms, such as BFS and DFS, can be used. At the beginning of the process, the model checker takes an initial predicate `Init`, a next-state action `Next`, a set of variables `vars` and a temporal property to be checked `Temporal`. First, a set of initial states are computed by evaluating the initial predicate `Init`. When executing BFS or DFS, neighbours of a node, that is the successor states of a known state, are computed by evaluating `Next`. For each of the states encountered, TLC checks whether the state violates any prescribed invariant. For each of the edge encountered, TLC checks whether the given temporal property is violated. If a property is violated, TLC produces a counterexample in the form of a path, that is a sequence of states in the directed graph. Just like the termination conditions of the standard BFS or DFS procedure, the model checking algorithm terminates when all reachable states are explored.

The model checking process is extremely resource-hungry, as the number of possible states are exponential in the number of variables. In addition, it is worth noting that the results obtained from the model checking process is as good as the model we use. In other words, whether we can faithfully model the real-world system

determines the quality of the results.

2.4 Summary

In this chapter, I talked about the logic system and concepts related to TLA. In Chapter 3, I will show how different constructs of TLA can be used to model concurrent data structures.

Chapter 3

Concurrent Data Structures

In Chapter 2, I gave an overview of relevant concepts of TLA. In this chapter, I will show how different components of concurrent programs and data structures can be modelled in TLA. I will also talk about how to express desired properties of systems using temporal formulae. Although there are many concurrent paradigms, such as messaging passing, this work focuses on concurrent data structures via shared memory. However, the idea behind can be adapted to other paradigms as well.

Some of the ideas in this chapter are either originated from or inspired by how PlusCal [Lamport, 2009] is translated to TLA+. In fact, there was an attempt of checking multithreaded data structures with PlusCal [Lamport, 2006].

3.1 Global and Local States

States (or variables) maintained by a program can be divided into two parts, global and local states. To model states, functions in TLA can be used. It is worth noting that functions in TLA carry a different meaning compared with functions in traditional programming languages. In TLA, functions are essentially mappings and work like dictionaries. For example, a function `foo` that maps each element in set A by applying operator `bar` can be defined like this `foo == [a \in A |-> bar(a)]`.

Each process accesses the shared memory by reads and writes of global variables. These variables map cleanly to variables in TLA, which can be changed by actions of all processes. An alternative way to model global variables is to establish a single TLA function that maps addresses to values `memory == [a \in Adr |-> 0]`. Then, we can assign a unique address to each global variable, just like how addressing works on real hardware.

On contrast, local states are only supposed to be accessible by the process that owns them. Therefore, a plain variable in TLA does not suffice, as it cannot maintain separate values for each of the processes. Note that although local states can be stored in registers or on the stack, we do not need to differentiate them in TLA.

A straightforward way to model a local state is by using a function mapping the ID of processes to values. For example, a local integer variable x initializing can be defined like this `x == [p \in PIDs |-> 0]`. To read a local state, we can simply query the function as follows `x[p]`. Similarly, to update a local state, we obtain a new

function by making it returning the new value for a particular process. Recall that actions represent relations between old states and new states. To increment *local state x*, an action needs to establish what *function* `x` looks like in the old and new states. It can be written as follows `x' = [x EXCEPT ![p] = x[p] + 1]`.

3.2 Control Flow

After encoding global and local states of programs using functions in TLA, we can easily map each statement of a program to an action in TLA. However, simply composing actions into the next-state action `Next` does not necessarily reflect the original program faithfully.

Suppose we have a program that first increments `x` and then assigns double of the value of `x` to `y`. The increment can be translated to action `x' = x + 1` and the assignment can be mapped to action `y' = 2 * x`. However, if we make `Next` a disjunction of the two above actions, the original order will not be maintained. For example, if we start with `x = 1 /\ y = 0`, we can end up in `x = 2 /\ y = 2` or `x = 2 /\ y = 4`, with `x = 1 /\ y = 2` and `x = 2 /\ y = 0` being the respective intermediate states. Note that both of the intermediate states are valid `Next`-steps with respect to the initial state.

One way to solve the problem is to borrow the idea of program counter (PC) or instruction pointer (IP). CPU uses PC to keep track of the instruction being executed. When there is a change in the control flow, PC will have been modified and CPU will load and execute next instruction in the new location. Analogously, we can maintain a special local state PC for each process. In addition, we assign a distinct "address" for each action in our program.

The control flow of a program can then be represented by the constraints of PC for each action. Each action is amended to require that the current PC matches the address of the action and the next PC matches the address of the next action. So for the above sequential example above, we use `I1` and `I2` as the addresses for the two actions respectively, and the next-state action of the program can be constructed as shown in Fig. 3.1. Branching behaviours can be modelled in the similar way, with the

```
1 I1(p) == /\ pc[p] = "I1"
2          /\ pc' = [pc EXCEPT ![p] = "I2"]
3          /\ x' = x + 1
4 I2(p) == /\ pc[p] = "I2"
5          /\ pc' = [pc EXCEPT ![p] = "Done"]
6          /\ y' = x + 1
7 Next(p) == I1(p) \/ I2(p)
```

Figure 3.1: The next-state action of `x += 1; y := 2*x`

address of next action being the branching target.

3.3 Modelling Concurrent Stacks and Queues

In this section, I will use two examples to show some tricky parts of modelling concurrent data structures in TLA. The Treiber stack [Treiber, 1986] is a classic non-blocking concurrent stack. It organizes the stack as a singly linked list, and each push and pop operation is done via atomic compare-and-swap operations on the head of the list. Treiber also proposed a non-blocking concurrent queue. However, it is inefficient, as dequeuing takes time proportional to the size of the queue. Michael and Scott [1996] proposed a non-blocking queue that has better performance.

In terms of modelling, the Treiber stack and the Michael-Scott queue share some similarities. Therefore, I will discuss them in the same section.

3.3.1 ABA Problem

The ABA problem commonly refers to synchronization problems where it is unsound to assume that an unchanged value in memory implies that nothing has changed.

A classic example would be a linked list that consists of `A -> B -> nil`. Thread 1 tries to pop `A`, and reads the head of the list. Then, thread 1 is swapped out and thread 2 is scheduled. Next, thread 2 pops both `A` and `B`, and pushes `A` back, resulting in `A -> nil`. Thread 2 subsequently frees the memory associated with `B`. After thread 1 resumes, it compares the head of the list, which is still A, and happily continues the pop operation by setting the head of the list to `B`, which is now invalid.

A common method to solve the problem is to conceptually associate a version number with a pointer, known as tagging. By doing this, a thread will be able to tell that something has changed even though the value might still be the same. There are many different approaches to implementing this efficiently on different hardware. Fortunately, in TLA, tagging can be modelled easily. Instead of using a mapping from addresses to values, the memory can be modelled as a mapping from addresses to tuples, where the first element of a tuple is the value and the second element is the version number (or timestamp). Whenever a value is modified (*e.g.*, the head of a list), the version number is incremented. In this way, even if a memory location is changed multiple times and back to its original value, a thread will be able to tell the difference as the version number will have changed.

3.3.2 Allocating New List Nodes

Both the Treiber stack and the Michael-Scott queue use linked lists to implement the underlying data structure. When inserting a new element, a new list node must be created and stored at some memory location. The actual implementations (*e.g.*, in C) usually allocate through the operating system or the allocation mechanism of the programming language (such as `malloc` or `new`). TLA does not explicitly support these operations, so I created a bare-minimum allocator.

A conceptually easy way is to use bump-pointer allocation. That is, we start with a chunk of free memory and allocate nodes contiguously. However, one important

drawback of this approach is the excessive memory consumption. Without the counterpart of garbage collection mechanisms, we cannot reclaim the memory of the nodes that are not in use. For example, if a process enqueues and dequeues alternatively, the memory required is proportional to the number of operations, even though the size of the queue stays constant. Recall that the model checking process takes time exponential in the number of variables. This might make the verification intractable.

Another approach is to use the idea of free-list allocation. The trick is to use another Treiber stack storing nodes that are available to use. Initially, all free memory is available in this stack. When allocating a node for either the Treiber stack or the Michael-Scott queue, we pop a node off the free stack. Similarly, when deallocating a node, we push this now-free node back into the free stack. In this way, we are able to cap the total number of addresses used and speed up the verification.

3.3.3 Specifying Properties

To verify whether the concurrent data structure is correct, it is important that we are able to specify desired properties. They can be safety properties, *i.e.*, something that should not be violated. They can also be liveness requirements, such as a process should make progress enqueuing a value. In terms of their abstraction levels, properties can be divided into two parts, user-visible values and internal consistency.

Users observe and manipulate data structures by issuing operations and examining return values. Unfortunately, TLA does not provide means to model function calls as in traditional programming languages. However, it is possible to have a workaround. The key observation is that when a function returns, the return value is local to the caller. That is, we can treat return values of a function as a special part of the local state. For example, if a stack only consists of positive values, we can have the following property (see Fig. 3.2) where `x` is the address of the action (see Section 3.2) immediately after a pop operation returns.

```
1 [](\A p \in processes: pc[p] = x => pop_return[p] > 0)
```

Figure 3.2: A property that a stack only consists of positive values. `[]` means *always* (□) and `\A` means *forall* (∀).

It is also important that the data structure is internally consistent. For example, at any point of time, we should be able to traverse from the head of the list, follow pointers, and reach a node pointed to by the tail of the list. I address the modelling of internal consistency by using the idea of *linearizability* from Herlihy and Wing [1990]. Linearizability requires that each operation seem to take effect instantaneously at some point between the invocation and response. In my modelling of the Michael-Scott queue, a variable `shadow_queue` is maintained using a sequence in TLA. When a process makes an operation visible by performing a compare-and-swap operation on the head/tail of the queue, the shadow queue is modified accordingly as shown in Fig. 3.3. By doing this, I am able to state a safety property shown in Fig. 3.4 that the internal linked list of the queue matches the shadow queue at any point of time.

```
def dequeue(q) {
    local head = load(q.head);
    ...
    if compare_and_swap(q.head, head, head.next) { // successful update
        shadow_queue = Tail(shadow_queue);
    }
    ...
}
```

Figure 3.3: The pseudocode of maintaining the shadow queue whiling dequeuing values

```
def compare(head, shadow_queue) {
    match head with
      | null => shadow_queue = <<>>
      | otherwise =>
          /\ Len(shadow_queue) /= 0
          /\ head.value = Head(shadow_queue)
          /\ compare(head.next, Tail(shadow_queue))
}
```

Figure 3.4: A property that the linked list matches the shadow queue

3.4 Summary

In this chapter, I showed how constructs of concurrent data structures can be represented in TLA. In the next chapter, I will discuss memory models, why it matters to the verification of concurrent data structures and how I modelled it in TLA.

Chapter 4

Memory Models

In Chapter 3, I showed how concurrent data structures can be represented in TLA. Another integral part of the verification process is modelling memory of our hardware targets. I will show why failing to take them into account can severely affect our reasoning about the correctness of programs. I will also show how these memory models can be encoded in TLA, which enables seamless integration with the TLA modelling of concurrent data structures.

4.1 Overview

One might imagine that the techniques presented in the previous chapter were sufficient to model and verify common concurrent data structures. Unfortunately, this is not the case if we care about the behaviours of these data structures on many modern, real-world hardware.

In Section 3.1, I use functions in TLA to model memory shared among processes, where a memory read/write corresponds to an action that queries/updates the corresponding TLA function. Recall that an action represents the change to program states in a step. Therefore, under this modelling, a memory operation loads from or stores to the atomic memory instantaneously (see Fig. 4.1). This model of memory is known

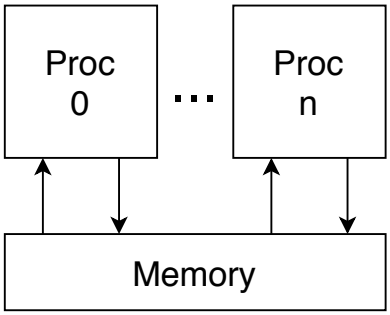

Figure 4.1: Abstract Model of SC

as *Sequential Consistency* (SC), where "the result of an execution is the same as if the operations had been executed in the order specified by the program" [Lamport, 1979].

In SC, all processors have the same view of the memory, and stores from a processor are visible to all processors at the same time.

Despite being definitionally simple, SC is not widely implemented on most hardware. Since SC prescribes such a strong order, many architectural optimizations (*e.g.*, store buffers) are forbidden, and therefore performance suffers. With the development of complex cache hierarchy and speculative execution on modern chips, it is increasingly expensive to maintain SC guarantees. Since the strong guarantees provided by SC are not necessary in many cases, hardware vendors deployed *weak memory architectures*. These architectures relax some constraints of SC in order to improve the performance of memory operations. To recover strong orders where necessary, these architectures also provide *memory fences* (or memory barriers) that prevent reordering. For example, x86 provides `LFENCE`, `MFENCE` and `SFENCE`.

Later, the behaviours of these architectures are formalized as *weak memory models*. For example, a weak memory model might allow reordering a store-load pair. That is, a load that happens after the store in program order might not see the new value being stored. This permits store buffering and processors do not need to block for a store to complete, which might lead to better performance.

However, the implications of relaxed orderings are profound. For example, suppose there are two processes relying on a shared variable for mutual exclusion. When one process enters the critical section, it atomically checks and changes the shared variable to indicate its possession of the lock. Then, when the other process checks the shared variable, it might see the stale value (*i.e.*, unlocked) due to store buffering, and subsequently enters the critical section as well. This leads to the violation of mutual exclusion.

The example above shows that not accounting for memory models leads to correctness issues even for such a simple concurrent data structure. Therefore, it is important that we integrate memory models into the verification process.

4.2 Litmus Tests

Just like shibboleths distinguish one group of people from another, litmus tests [Alglave et al., 2011] are quite effective in revealing the differences between different memory models. Each litmus test consists a multithreaded program, and assertions regarding shared memory and local states of each thread. These assertions can be something that is required, possible, or forbidden. Litmus tests are used in the following sections to help the reader better understand possible behaviours introduced by weak memory models.

Litmus tests are also helpful for me to establish the correctness of my encoding of memory models. Using the encodings shown in Chapter 3, I can systematically generate TLA models from these tests using templates. By comparing observable outcomes with known models, I can determine whether my encoding allows desirable behaviours and disallows erroneous outcomes.

In the remaining parts of this chapter, I will discuss what the implications of

various memory models are, and how their operational models can be realized in TLA.

4.3 Semantics of Memory Models

Before encoding memory models, we should first understand what their implications are. Formally, memory models define rules about loads and stores, and how they act upon memory [Sorin et al., 2011]. In other words, among many possible outcomes due to reordering and interleaving, a memory model allows correct execution results and disallows (many more) illegal ones.

As shown by Sewell et al. [2010], informal prose is not a good medium for specifying memory models. These loose descriptions do not address some of the subtleties and are inevitably unsound. Worse still, one cannot check real programs against such loose specifications, which leads to confusion when reasoning about executions involving multiple threads. To address these problems, two formalizations of memory models are commonly used instead in the literature. They are *axiomatic semantics* and *operational semantics*.

An axiomatic semantics makes assertions on the relations of memory operations [Alglave et al., 2014]. For example, suppose there is a memory model that forbids load-store reordering. Then for a pair $(\mathrm{Ld}, \mathrm{St})$, if the load precedes the store in program order, their memory order should also be maintained in a valid execution. However, this approach relies on guessing the entire execution trace of a program, which is prohibitive for large programs that involve loops, branches, etc. [Zhang et al., 2017].

An operational semantics describes computation using execution rules on an abstract machine. This allows us to derive possible results of program execution mechanically. *Instantaneous Instruction Execution* ($\mathrm{I}^2\mathrm{E}$) is a popular notation used in [Sewell et al., 2010] and [Zhang et al., 2017]. In $\mathrm{I}^2\mathrm{E}$, possible reorderings of instructions are captured by putting different types of buffers between processors and the atomic memory. $\mathrm{I}^2\mathrm{E}$ will be used in the following sections to describe the operational semantics of some memory models.

The following two sections discuss the design and implementation of TSO and WMM in TLA. By encoding memory models in the same logic used to encode programs, I enable seamless integration. That is, by replacing memory reads and writes with the corresponding operations in TSO or WMM, we can observe possible executions of a program after relaxing some ordering constraints. To make it easier for the reader to follow, pseudocode based on PlusCal syntax is used instead of raw TLA+.

4.4 Total Store Ordering

Total Store Ordering (TSO) is a widely implemented memory model, notably on x86 and SPARC architectures. Compared with SC, the most notable difference of TSO is that it permits store-load reordering.

To illustrate the implication, the following litmus test (see Fig. 4.2) is used. No

Proc 0	Proc 1
`St x 1`	`St y 1`
`r0 = Ld y`	`r1 = Ld x`
Proc 0: `r0 = 0` $\wedge$ Proc 1: `r1 = 0` allowed on TSO	

Figure 4.2: SB: test for store buffer

interleaving of instructions can lead to the above behaviour. However, due to store-load reordering, the reads of `x` and `y` can be reordered before the writes, which results in reading the stale values.

In the operational model due to Sewell et al. [2010], *store buffers* (SB) are introduced between each process and the atomic memory (see Fig. 4.3) to model store-load reordering. Since there is a store buffer associated with each processor, store buffers

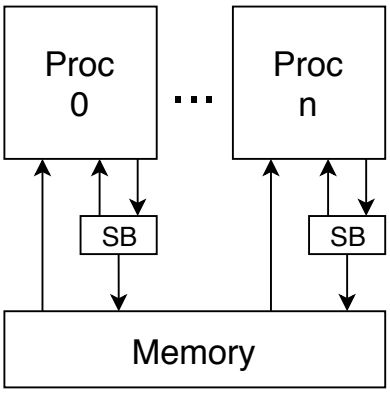

Figure 4.3: Abstract Model of TSO

can be treated as a special part of the local state and the FIFO buffer is represented by a sequence in TLA. Instead of interacting with the atomic memory directly, stores from a processor are queued into the respective buffer (see Fig. 4.4). When a processor

```
1 def store(addr, val) {
2     store_buffer[pid] := Append(store_buffer[pid], <<addr, val>>);
3 }
```

Figure 4.4: The store operation of TSO

loads from a memory address (see Fig. 4.5), it queries its own store buffer first,

returning the freshest store to that address if found. Otherwise, the processor loads from the memory. Concurrently, a background process (see Fig. 4.6) repeatedly,

```
def load(addr) {
    // To load the freshest store, we start from the tail of the buffer
    return load_aux(addr, Len(store_buffer[pid]));
}

def load_aux(addr, idx) {
    if idx = 0 {
        return memory[addr];
    } else {
        if store_buffer[pid][idx][1] = addr { // entries are address, value pairs
            return store_buffer[pid][idx][2];
        } else {
            return load_aux(addr, idx - 1);
        }
    }
}
```

Figure 4.5: The load operation of TSO

nondeterministically chooses a processor, dequeues the oldest entry in its store buffer, and updates the memory[1]. To recover SC behaviour, a `commit` fence can be inserted

```
proc background {
    while TRUE do
        with p \in Procs do
            if Len(store_buffer[p]) /= 0 then
                head := Head(store_buffer[p]);
                store_buffer[p] := Tail(store_buffer[p]);
                memory[head[1]] := head[2];
            end if;
        end with;
    end while;
}
```

Figure 4.6: The background process of TSO

after a store, which blocks the processor until its store buffer is empty (see Fig. 4.7). This ensures all stores preceding the fence are visible to other processors.

I ran the litmus tests SB, IRIW, n6, n5, n4b, example 8-1, example 8-2, example 8-4, example 8-6, example 8-9, example 8-10 and AMD5 as described in [Sewell et al., 2010]. The results I obtained matches the description in the literature. As we will see in the next section, the implementation of TSO acts as a stepping stone towards the implementation of WMM, as their operational models share some similarities.

[1] `with` represents nondeterministic choices among the members of a set

```
1 def commit() {
2     while Len(store_buffer[self]) /= 0 {
3     }
4     return;
5 }
```

Figure 4.7: The commit fence of TSO

4.5 WMM

It would also be interesting to study and encode some weaker memory models compared with TSO. Such memory models can be found in commercial architectures, such as ARM and POWER. However, their axiomatic and operational semantics are very complex [Sarkar et al., 2011; Flur et al., 2016]. In addition, both of the operational models rely on maintaining out-of-order (OOO) thread subsystems to permit all possible behaviours. The thread subsystems keep track of instructions committed/in-flight, where in-flight instructions are subjected to restarting if the speculation turns out to be unsound. This requires the model to maintain some sort of *reorder buffer* (ROB), which complicates the implementation.

WMM was proposed by Zhang et al. [2017] as a definitionally simple yet flexible memory model for RISC-V. It permits all reorderings except load-store reordering. This trade-off does not hinder the performance much, as per Zhang et al.'s performance evaluation. In addition, it does not enforce any dependency ordering, including data-dependent loads and control flow dependency. To demonstrate the implications, I use two litmus tests. As shown in Fig. 4.8, stores cannot overtake loads, and it cannot be the case that both `r1` and `r2` are 1. WMM does allow other

Proc 1	Proc 2
I_1: `r1 = Ld b`	I_3: `r2 = Ld a`
I_2: `St a 1`	I_4: `St b 1`
`r1 = r2 = 1` forbidden on WMM	

Figure 4.8: LB: test for load-store reordering

reorderings, such as load-load reordering, as shown in Fig. 4.9. Due to the insertion of a full fence (which prevents instructions moving across the fence) at I_2, the two stores from processor 1 must update memory in order. However, processor 2 is permitted to see an up-to-date version of `b` and a stale version of `a`, despite the presence of control flow dependency at I_5.

The operational semantics of WMM is also given in I^{2}E, making it a natural next step from the encoding for TSO. Possible behaviours of WMM are captured nicely using two kinds of buffers, which leads to clean definition and implementation. In addition to store buffers (as in TSO), WMM introduces *invalidation buffers* (IB) for each processor (see Fig. 4.10). The invalidation buffer, which is address-segregated,

Proc 1	Proc 2
I_1: `St a 1`	I_4: `r1 = Ld b`
I_2: `FENCE`	I_5: `if (r1 != 0) exit`
I_3: `St b 1`	I_6: `r2 = Ld a`
`r1 = 1, r2 = 0` allowed on WMM	

Figure 4.9: MP+Ctrl: test for control dependency ordering

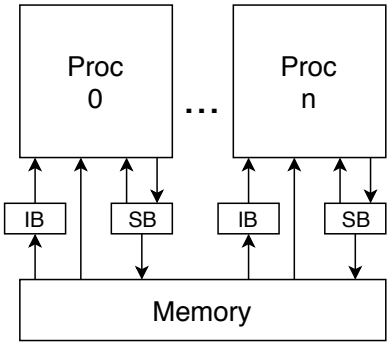

Figure 4.10: Abstract Model of WMM

allows a processor to see stale versions of an address. But once the processor has seen a value v from address a at time t, it cannot see values from times before t. The store buffers in WMM have a new flavour. Recall that in TSO, entries in the store buffer for a processor are ordered by time, regardless of the address, and hence the name *Total Store Ordering*. In WMM, store buffers are address segregated, just like invalidation buffers. This allows stores of different addresses to update memory in different order. These buffers lead to the behaviours shown in the litmus tests above.

Luckily, the encoding of WMM in TLA is not much harder if appropriate design choices are made. Since both buffers are address-segregated, the trick is to make buffers mappings from addresses to tuples for each processor (see Fig. 4.11). I will

```
1 variables sb = [p \in Procs |-> [a \in Adr |-> <<>>]],
2           ib = [p \in Procs |-> [a \in Adr |-> <<>>]];
```

Figure 4.11: Address-segregated definition of buffers of WMM

show how this representation helps the modelling of operations, starting with stores. A store puts the new value into the corresponding SB and flushes the corresponding IB as shown in Fig. 4.12. Concurrently, a background process repeatedly, nondeterministically chooses a processor and an address (see Fig. 4.13). If the corresponding SB is non-empty, it dequeues the oldest entry (WMM still maintains the store ordering for an address), and updates the memory. In addition, the old memory value for that address is added to the IB of every other process whose SB does not contain the

```
def store(addr, val) {
    sb[pid][addr] := Append(sb[pid][addr], val);
    ib[pid][addr] := <<>>;
}
```

Figure 4.12: The store operation of WMM

address. Finally, a load operation as shown in Fig. 4.14 checks whether an address

```
proc background {
    while TRUE do
        with p \in Procs do
            with a \in Adr do
                if Len(sb[p][a]) /= 0 then
                    // find the oldest entry for an address in sb
                    head := Head(sb[p][a]);
                    // for all processes
                    ib := [p' \in Procs |->
                        // leave the ib of the original process untouched
                        // or if sb contains the address
                        IF p' = p \/ Len(sb[p'][a]) = 0
                        THEN ib[p']
                        ELSE [ib[p'] EXCEPT ![a] = Append(ib[p'][a], memory[a])]
                    ];
                    sb[p][a] := Tail(sb[p][a]);
                    memory[a] := head;
                end if;
            end with;
        end with;
    end while;
}
```

Figure 4.13: The background process of WMM. `with` represents nondeterministic choices among the members of a set.

is present in SB. If that is the case, the store operation simply returns the freshest entry. Otherwise, it nondeterministically looks up the memory or IB. If it queries the memory, then the corresponding IB will be purged. If it reads an entry of the IB, then any older entry will be removed. Either way, once a processor sees a version of an address, it will not see any older version. WMM provides a `commit` fence as shown in Fig. 4.15, which can be implemented in a similar fashion compared with TSO. In addition, WMM provides a `reconcile` fence as shown in Fig. 4.16, which clears the IB of the issuing processor. This prevents the processor from reading stale values.

4.6 Dependency Ordering of Different Models

Usually, many microarchitectural details of processors are not visible to programmers modulo performance implications. However, some reorderings can lead to different and quite unexpected execution results. Especially with respect to dependency order-

```
def load(addr) {
    if Len(sb[pid][addr]) /= 0 then
        // address is presented in sb
        idx := Len(sb[pid][addr]);
        return sb[pid][addr][idx];
    else
        either
            val := memory[addr];
            ib[pid][addr] := <<>>;
            return val;
        or
            with i \in 1..Len(ib[pid][addr]) do
                val := ib[pid][addr][i];
                // purging older entries
                ib[pid][addr] := ib[pid][addr][i..Len(ib[pid][addr])];
                return val;
            end with;
        end either;
    end if;
}
```

Figure 4.14: The load operation of WMM. `either ... or ...` represents nondeterministic choices.

```
def commit() {
    while (\E a \in Adr: Len(sb[self][a]) /= 0) {
    }
    return;
}
```

Figure 4.15: The commit fence of WMM. `\E` means *exists* (∃).

```
def reconcile() {
    ib[pid] := [a \in Adr |-> <<>>];
    return;
}
```

Figure 4.16: The reconcile fence of WMM

ing, different memory models show a wide spectrum of behaviours. This warrants some summary here.

Two litmus tests, PPO015 ("PPO") and MP+dmb+fri+rfi-ctrlisb ("MP"), are used here. These tests demonstrate different behaviours exhibited on different memory models with regard to control-flow dependency and data dependency. In litmus test PPO (see Fig. 4.17), the store in I_5 has a data dependency on the load in I_4 (even though `r0 xor r0` always evaluates to 0). Therefore, I_7 depends on I_4 via `z`. In litmus

Proc 1	Proc 2
I_1: `St x 1`	I_4: `r0 = Ld y`
I_2: `FENCE`	I_5: `St z (r0 xor r0)+1`
I_3: `St y 1`	I_6: `St z 2`
	I_7: `r3 = Ld z`
	I_8: `if r3 = r3 then skip else skip`
	I_9: `CFENCE`
	I_{10}: `r4 = Ld x`
`r0 = 1, r4 = 0?`	

Figure 4.17: Litmus test PPO015

test MP (see Fig. 4.18), the data dependency is removed, and I_6 directly depends on I_4 via `y`. Both tests have a control fence[2] (`CFENCE`) inserted. These fences are

Proc 1	Proc 2
I_1: `St x 1`	I_4: `r0 = Ld y`
I_2: `FENCE`	I_5: `St y 2`
I_3: `St y 1`	I_6: `r3 = Ld y`
	I_7: `if r3 = r3 then skip else skip`
	I_8: `CFENCE`
	I_9: `r4 = Ld x`
`r0 = 1, r3 = 2, r4 = 0, x = 1, y = 2?`	

Figure 4.18: Litmus test MP+dmb+fri+rfi-ctrlisb

meant to prevent loads (I_{10} of PP015 and I_9 of MP) from happening before the branch instructions (I_8 of PPO and I_7 of MP). On the face of it, the control fence together with the data dependency should keep the load and the branching in order. However, as indicated by the results below, these conditions are not sufficient in some memory models. The results are summarized in Table 4.1.

The model proposed by Colvin and Smith [2018] is based on pair-wise reordering. It was designed to be a generic framework for verifying programs on weak memory

[2]Control fences are implemented as `isync` on POWER, `isb` on ARM and `reconcile` on WMM

Memory Model	PPO		MP	
	Hardware	Model	Hardware	Model
Colvin and Smith	N/A	✓	N/A	✓
POWER	?	✓	×	×
ARM	×	×	✓	✓
WMM	N/A	×	N/A	×

Table 4.1: Results of PPO015 and MP+dmb+fri+rfi-ctrlisb on different memory models. ✓ means the outcome is allowed by the model or observable on machines, × means the outcome is forbidden by the model or not observable on machines, and ? means the outcome is not reported by the literature I surveyed.

models, and was instantiated to capture the behaviours of the ARM and POWER processors. Both of these litmus tests are permitted by this model.

The operational model of ARMv8 due to Flur et al. [2016] permits MP, and the behaviour is observable on some ARM machines. However, this model forbids PPO due to the data dependency.

The operational model of POWER due to Sarkar et al. [2011] forbids MP, and it is not observed on tested POWER machines [Alglave et al., 2014]. The model permits PPO, but the literature I have surveyed does not report whether this behaviour is observable on POWER machines. I could not test this myself as I do not have access to POWER machines.

WMM forbids both litmus tests. From the point of view of the axiomatic model, `reconcile` orders all instructions. That is, all instructions after the fence in program order also come after the fence in memory order. From the point of view of the operational model, `reconcile` purges the IB, and the subsequent load will not read the stale value.

4.7 Summary

In this chapter, I gave an overview of memory models and their semantics. I demonstrated how TSO and WMM can be implemented in TLA, and therefore be integrated as a part of the verification. I also discussed some discrepancies between different memory models with respect to dependency ordering. In the next chapter, I will use the Chase-Lev queue as a concrete example to show these encodings can be used to verify complex, real-world data structures.

Chapter 5

Case Study: the Chase-Lev Queue

In Chapter 3 and Chapter 4, I showed how different pieces of software systems can be represented in TLA. In this Chapter, I will use the Chase-Lev queue under the WMM memory model as a concrete example to show how TLA can be used to model real-world data structures.

5.1 Overview

The generic workflow of verifying a concurrent data structure is shown in Fig. 5.1. First, the program and the corresponding memory model need to be translated into

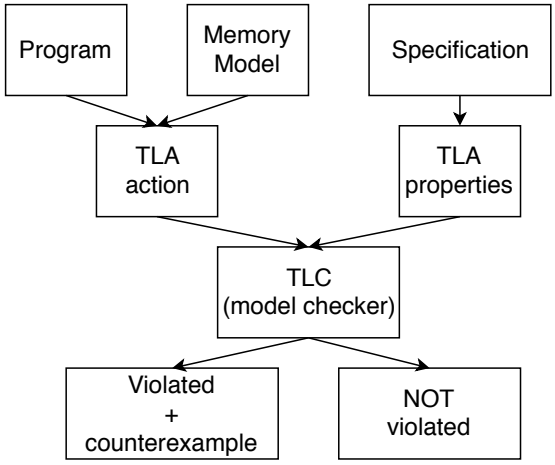

Figure 5.1: The workflow of verifying a program

a TLA next-step action. Then, the specifications need to be translated into TLA properties using temporal formulae. Next, the model checker takes the TLA action and properties as inputs, and explores reachable states. If any property is violated, the model checker will report the violation with a counterexample.

In previous chapters, I have addressed parts of the workflow. In Chapter 3, I showed how a concurrent data structure and its specifications can be expressed in TLA in general. And in Chapter 4, I demonstrated how memory models can be encoded in TLA. In this chapter, I will show how these can be combined to achieve a

verification goal for any given program. In particular, I will use a C implementation of the Chase-Lev queue as an example to show how the above steps are realized.

5.2 The Chase-Lev Queue

The Chase-Lev queue [Chase and Lev, 2005] is a work-stealing queue using a circular array as the backing storage. It is an important data structure for load balancing between parallel workers. Each worker owns a queue of work items, and the worker generates or consumes work by pushing and popping at the bottom of the queue. When a worker runs out of work, it may steal from the top (instead of the bottom to minimize contention) of the queues of other workers.

This data structure has also attracted some research interest in recent years. Lê et al. [2013] provided implementations for weak memory models in C and ARMv7. The C implementation uses atomic operations provided by the C11 standard [ISO, 2011]. The ARM implementation is hand-tuned to use as few fences as possible, and Lê et al. claimed to have proven its correctness against [Sarkar et al., 2011]. However, as pointed out by the literature [Norris and Demsky, 2013; Ou and Demsky, 2017; Colvin and Smith, 2018], both the C and ARM implementations contain bugs.

It would be nice that if I could reproduce these results in TLA to validate these claims in prior work.

5.3 Bug in the C implementation

5.3.1 Mapping C Atomic Operations

The excerpt of the C11 implementation of the Chase-Lev queue provided by Lê et al. is shown in Fig. 5.2. The first step of the verification process is to translate the C11 atomic operations into WMM operations. Fortunately, the translation is readily available in [Zhang et al., 2017]. Specifically, loads with `acquire` translate to `load; reconcile` sequences. This is to prevent instructions after the load overtaking it. That is, subsequent instructions should not see stale values via IB. Stores with `release` translate to `commit; store` sequences. This it to ensure the store not overtake previous instructions. That is, previous instructions should successfully update the memory before executing the current instruction.

The translation of `compare_exchange` is discussed below.

5.3.2 Implementing Compare-Exchange

The steal operation uses a compare-exchange operation for correct synchronization on the top of the queue, and it is important that I implement this operation. However, implementing compare-exchange for the WMM model in TLA proves to be surprisingly complicated. The most important and difficult part is to ensure the atomicity of the operation. Recall that the atomicity is only maintained at the action-level in TLA. Since each operation consists of multiple TLA actions, I adopted a lock, which will

```
void push(Deque *q, int x) {
    size_t b = load_explicit(&q->bottom, relaxed);
    size_t t = load_explicit(&q->top, acquire);
    Array *a = load_explicit(&q->array, relaxed);
    if (b - t > a->size - 1)
        resize(q);
    store_explicit(&a->buffer[b % a->size], x, relaxed);
    thread_fence(release);
    store_explicit(&q->bottom, b + 1, relaxed);
}
int steal(Deque *q) {
    size_t t = load_explicit(&q->top, acquire);
    thread_fence(seq_cst);
    size_t b = load_explicit(&q->bottom, acquire);
    int x = EMPTY;
    if (t < b) {
        Array *a = load_explicit(&q->array, relaxed);
        x = load_explicit(&a->buffer[t % a -> size], relaxed);
        if (!compare_exchange_strong_explicit(&q->top, &t, t+1, seq_cst, relaxed))
            return ABORT;
    }
    return x;
}
```

Figure 5.2: The C11 implementation of Chase-Lev queue

be explained below. This seems paradoxical as people use atomic operations, such as compare-exchange, to avoid using locks. However, a lock is essential in a TLA modelling to ensure the processor issuing the operation has exclusive access to the memory.

A compare-exchange operation with `seq_cst` ordering is decomposed into the following steps. The process first atomically checks and acquires the lock. If it successfully acquires the lock, it also sets the lock owner to itself simultaneously (see Fig. 5.3). Then, it issues a `commit; reconcile` sequence to make sure that all

```
def lock() {
    if Lock /= LOCKED then
        Lock := LOCKED;
        Lock_Owner := pid;
    end if;
}
```

Figure 5.3: The lock operation of WMM

previous stores are committed to the memory, and it sees an up-to-date version of the memory. Then, it loads the memory, and checks whether the value is expected. If that is the case, it stores to the memory, and then issues a `commit` fence to make sure the update can be seen by any subsequent instructions. After completing all these actions, it releases the lock (see Fig. 5.4).

With the presence of a lock, any operation that reads or writes the memory also

```
1  def unlock() {
2      assert Lock = LOCKED /\ Lock_Owner = pid;
3      Lock := UNLOCKED;
4      Lock_Owner := nil;
5  }
```

Figure 5.4: The unlock operation of WMM

needs to be modified accordingly. Specifically, the background process and the load need to be changed. The background process can only dequeue an address-value pair from the SB of the lock owner when the lock is held (see Fig. 5.5). Note that the store

```
1  proc background {
2      while TRUE do
3          with p \in {p' \in Procs: Lock = UNLOCKED \/ Lock_Owner = p'} do
4              ...
5          end with;
6      end while;
7  }
```

Figure 5.5: The background process of WMM with the presence of lock

operations do not need to be changed as they only affect the SB of the issuing process, and the modified version of the background process ensures that stores from other processes will not update the memory while the lock is held. The load operations simply block when the lock is held by other processes as shown in Fig. 5.6.

```
1  def load(addr) {
2  _loop:
3      if Lock = LOCKED /\ Lock_Owner /= self then
4          goto _loop;
5      else
6          ...
7      end if;
8  }
```

Figure 5.6: The load operation of WMM with the presence of lock

This completes the encoding of compare-exchange operations of `seq_cst` ordering.

5.3.3 Checking Properties

As pointed out by Norris and Demsky [2013], the bug in the C implementation is that the ordering at line 17 of Fig. 5.2 is `relaxed`, which is too weak. The load of queue data at line 18 can therefore be reordered before the load of array. This can result in reading uninitialized memory when the queue is resized and moved.

To verify this claim, I initialized all memory addresses to a special value, *e.g.*, 42. I defined two processes, one for enqueuing and another for stealing. Their PIDs are 1 and 2 respectively. Process 1 tries to steal from the queue. Process 2 first pushes 2 to the queue and copies the buffer to a new location, mimicking resizing of the queue. I defined a safety property that process 1 should never read uninitialized memory, *i.e.*, the return value of `steal` should never be 42.

I ran the model checker, which took around 11 seconds with 4 worker threads on a laptop with Intel® Core™ i5-5257U CPU. Note that this is not intended to be a rigorous performance evaluation. It is meant to give the reader a rough idea about the performance of the TLA toolchain. The model checker successfully reported the violation of the property. Since the load of array is `relaxed`, there is no insertion of a memory fence. As a result, the subsequent load of queue data could and did read the stale value from the IB. To fix the bug, the load of array needs to be changed to `acquire` ordering. This will insert a `reconcile` fence after the load of array, which will purge the IB and prevent the read of uninitialized memory. This is consistent with what was reported by Norris and Demsky.

5.4 Discussion

In Section 5.3, I use the C implementation of the Chase-Lev queue as an example to show the workflow of using TLA to verify a concurrent data structure. The idea behind, however, is not limit to this particular data structure or programming language. In fact, one advantage of using TLA is the elegance of unified representation. Since the program, the memory model and the specification are all written in the same logic, they can be reused across different verification goals. To handle a different programming language, only the translation from that language to TLA needs to be changed, and all programs in that language can then be automatically translated. Likewise, to adapt a different memory model, one can keep programs untouched and only redefine how loads, stores and memory fences work. Last but not least, all of these can be checked using the same tool. The above properties make TLA an appealing candidate for verifying concurrent data structures.

The flexibility might come with the cost of performance. For example, being a generic tool, the current version of the model checker is unlikely to exploit the property of memory buffers and adapt a better internal representation. I argue that this is not an inherent problem with TLA, as it can be mitigated by providing more efficient primitive types and more runtime optimizations.

5.5 Bug in the ARM implementation

Colvin and Smith [2018] claimed that the ARM implementation given by Lê et al. contains an unnecessary control fence, and another incorrectly placed control fence. Since WMM and ARM do not have equivalent memory models, I cannot validate

the claim using WMM. I decided to consult the mapping of C atomic operations to ARM [Sewell, 2016].

Disregarding the bug discussed in the previous section, the ARM implementation still contains a bug in the `steal` operation. The C code was already shown in Fig. 5.2. The corresponding ARM code, also provided by Lê et al., is shown in Fig. 5.7. Since

```
1  int steal(Deque *q) {
2      size_t t = R(q->top);
3      sync;
4      size_t b = R(q->bottom);
5      ctrl_isync;
6      int x = EMPTY;
7      if (t < b) {
8          Array *a = R(q->array);
9          x = R(a->buffer[t % a -> size]);
10         ctrl_isync;
11         bool success = false;
12         atomic
13             if (success = (R(q->top) == t))
14                 W(q->top, t + 1);
15         if (!success)
16             return ABORT;
17     }
18     return x;
19 }
```

Figure 5.7: The ARM implementation of the steal operation

`relaxed` operations do not order anything, any insertion of a fence must be due to a stricter C ordering. It is clear that the full ARM fence `sync` inserted at line 3 of the ARM code corresponds to line 13 of the C code. Similarly, the control fence inserted at line 5 of the ARM code is meant to realize the `acquire` semantics of the load at line 14 of the C code. Preceded by two `relaxed` loads, the control fence at line 10 of the ARM code can only correspond to the compare-and-swap operation at line 19 of the C code.

However, the explanation (see Fig. 5.8) in [Colvin and Smith, 2018] does not seems to be faithful. Since they have assumed `CAS` as a primitive, the `cfence` at line 8 should not be included. However, the general reasoning in [Colvin and Smith, 2018] is correct: to prevent the speculative load of line 7 in Fig. 5.8, a `cfence` must be placed after the guard at line 6. That is, line 10 in the ARM code (see Fig. 5.7) should be moved after the conditional at line 7. This is to ensure that the `acquire` semantics of line 14 of the C code (see Fig. 5.2) is correctly realized. This is cross-checked by Sewell [2016], who suggests that a correct translation of load-acquire in C is `ldr; teq; beq; isb` to ensure the control-flow dependency[1].

[1] Will Deacon at ARM remarks that although this usage is sound, a `ldr; dmb ish` sequence is preferred [Sewell, 2016].

```
1  steal
2      h := head;
3      fence;
4      t := tail;
5      cfence;
6      if h < t then
7          return := tasks[h mod L];
8          cfence;
9          if !CAS(head, h, h+1) then
10             return := fail
11     else
12         return := empty
```

Figure 5.8: The translation of the ARM implementation of the steal operation

5.6 Summary

In this chapter, I showed the bug-finding process for an implementation of the Chase-Lev queue. Starting from source code in C11 atomic operations, I translated it into the corresponding WMM operations. Then, by specifying the property as an invariant, the model checker successfully found a problematic execution, which confirmed the bug found in prior work. I discussed that how the same idea can be adapted to other problems. I argued that TLA is indeed capable of verifying concurrent data structures on weak memory models. At the end of the chapter, I also discussed the bug in the ARM implementation of the Chase-Lev queue.

Chapter 6

Conclusion

Verifying computer programs of any considerable size is hard. With concurrent execution and the presence of weak memory models, the verification only becomes harder. Since manual verification is less tractable in this case, many different machine-aided approaches have been proposed. In this report, I argue that TLA and the related software toolchains are suitable for modelling and verifying concurrent data structures.

Chapter 2 summarizes TLA, and shows how this system's logic can be related to the program execution and properties.

In Chapter 3, I showed in general how concurrent data structures can be mapped to TLA constructs. By using functions in TLA, both the local and global state of a program can be captured. In addition, the control flow of programs can also be represented in TLA via the idea of program counters. Moreover, I used the Treiber stack and the Michael-Scott queue to demonstrate patterns involved in encoding concurrent data structures and their specification.

Then, in Chapter 4, I discussed memory models and their semantics. I showed how two weak memory models, TSO and WMM, can be smoothly encoded in TLA. This enables seamless integration with concurrent data structures expressed in TLA. Throughout the chapter, litmus tests are used to reveal the difference between memory models. These tests also help when I checked my encoding against models in prior work. At the end of the chapter, I discussed the impact of dependency ordering on different memory models, which show quite a wide range of behaviours.

Finally, Chapter 5 used the Chase-Lev queue as a concrete example to show how TLA can be applied to real-world data structures. Starting with C source code, I first translated it to WMM operations. After constructing a sample scenario (push and then resize) and specifying an invariant, I successfully found the bug found in prior work. The results show that TLA is appropriate for modelling and verifying concurrent data structures.

6.1 Future Work

More Comprehensive Litmus Tests Section 4.2 discussed litmus tests and how they help contrasting different memory models and establishing the correctness of

my encodings. However, I only used a small set of litmus tests due to limited time.

There is a collection of litmus tests available online[1]. Future work could try automatically translating these tests into their TLA encoding. This comprehensive set of tests can help iron out bugs in corner cases. It might also be helpful to generate more litmus tests automatically as discussed in [Wickerson et al., 2017; Mador-Haim et al., 2010]

Automatic Program Translation In Chapter 3, I showed how concurrent data structures in general can be expressed in TLA. And in Chapter 5, I showed how to translate operations of a programming language into operations of weak memory models. The above process is quite manual, yet systematic. This means that it would be worthwhile developing a toolchain that takes an assembly or C program and translates it into TLA. Such tools can make the verification process much more ergonomic and thus appeal to a broad audience.

Asynchronous Satisfaction of Memory Requests In this work, I only encoded the operational models of TSO and WMM in TLA. Their behaviours are nicely captured by buffers like store buffers and invalidation buffers. These buffers can be easily mapped to TLA using functions and sequences.

There are other operational models, such as Flowing/POP [Flur et al., 2016] that are based on the idea of satisfying memory requests. However, these require a thread subsystem that does explicit OOO, which is not easy to implement in TLA as it is not a generic programming language. Future work might model this in TLA by allowing asynchronous satisfaction of memory requests.

[1] http://www.cl.cam.ac.uk/users/pes20/rmem

Bibliography

Alglave, J.; Maranget, L.; Sarkar, S.; and Sewell, P., 2011. Litmus: Running tests against hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, TACAS'11/ETAPS'11 (Saarbrücken, Germany, 2011), 41–44. Springer-Verlag, Berlin, Heidelberg. http://dl.acm.org/citation.cfm?id=1987389.1987395. (cited on page 14)

Alglave, J.; Maranget, L.; and Tautschnig, M., 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36, 2 (Jul. 2014), 7:1–7:74. doi:10.1145/2627752. http://doi.acm.org/10.1145/2627752. (cited on pages 15 and 23)

Chase, D. and Lev, Y., 2005. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05 (Las Vegas, Nevada, USA, 2005), 21–28. ACM, New York, NY, USA. doi:10.1145/1073970.1073974. http://doi.acm.org/10.1145/1073970.1073974. (cited on page 26)

Chaudhuri, K.; Doligez, D.; Lamport, L.; and Merz, S., 2010. Verifying safety properties with the TLA + proof system. In *Automated Reasoning*, 142–148. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 5)

Colvin, R. J. and Smith, G., 2018. A wide-spectrum language for verification of programs on weak memory models. In *Formal Methods*, 240–257. Springer International Publishing, Cham. (cited on pages 22, 23, 26, 29, and 30)

Flur, S.; Gray, K. E.; Pulte, C.; Sarkar, S.; Sezgin, A.; Maranget, L.; Deacon, W.; and Sewell, P., 2016. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. *SIGPLAN Not.*, 51, 1 (Jan. 2016), 608–621. doi:10.1145/2914770.2837615. http://doi.acm.org/10.1145/2914770.2837615. (cited on pages 18, 23, and 34)

Herlihy, M. P. and Wing, J. M., 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12, 3 (Jul. 1990), 463–492. doi:10.1145/78969.78972. http://doi.acm.org/10.1145/78969.78972. (cited on page 10)

ISO, 2011. Information technology – programming languages – C. Standard, International Organization for Standardization, Geneva, CH. (cited on page 26)

Jones, C. B., 1983. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, 321–332. North-Holland. (cited on page 1)

Lamport, L., 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28, 9 (Sep. 1979), 690–691. doi:10.1109/TC.1979.1675439. https://doi.org/10.1109/TC.1979.1675439. (cited on page 13)

Lamport, L., 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16, 3 (May 1994), 872–923. doi:10.1145/177492.177726. http://doi.acm.org/10.1145/177492.177726. (cited on pages 1 and 3)

Lamport, L., 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 032114306X. (cited on page 5)

Lamport, L., 2006. Checking a multithreaded algorithm with $^{+}$CAL. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06 (Stockholm, Sweden, 2006), 151–163. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/11864219_11. http://dx.doi.org/10.1007/11864219_11. (cited on page 7)

Lamport, L., 2009. The PlusCal algorithm language. In *Theoretical Aspects of Computing - ICTAC 2009*, 36–60. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on pages 5 and 7)

Lê, N. M.; Pop, A.; Cohen, A.; and Zappa Nardelli, F., 2013. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13 (Shenzhen, China, 2013), 69–80. ACM, New York, NY, USA. doi:10.1145/2442516.2442524. http://doi.acm.org/10.1145/2442516.2442524. (cited on pages 26, 29, and 30)

Mador-Haim, S.; Alur, R.; and Martin, M. M. K., 2010. Generating litmus tests for contrasting memory consistency models. In *Computer Aided Verification*, 273–287. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 34)

Michael, M. M. and Scott, M. L., 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96 (Philadelphia, Pennsylvania, USA, 1996), 267–275. ACM, New York, NY, USA. doi:10.1145/248052.248106. http://doi.acm.org/10.1145/248052.248106. (cited on page 9)

Norris, B. and Demsky, B., 2013. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13 (Indianapolis, Indiana, USA, 2013), 131–150. ACM, New York, NY, USA. doi:10.1145/2509136.2509514. http://doi.acm.org/10.1145/2509136.2509514. (cited on pages 26, 28, and 29)

O'Hearn, P. W., 2007. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375, 1 (2007), 271 – 307. doi:https://doi.org/10.1016/j.tcs.2006.12.035. http://www.sciencedirect.com/science/article/pii/S030439750600925X. Festschrift for John C. Reynolds's 70th birthday. (cited on page 1)

Ou, P. and Demsky, B., 2017. Checking concurrent data structures under the C/C++11 memory model. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17 (Austin, Texas, USA, 2017), 45–59. ACM, New York, NY, USA. doi:10.1145/3018743.3018749. http://doi.acm.org/10.1145/3018743.3018749. (cited on page 26)

Pnueli, A., 1977. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, 46–57. IEEE Computer Society, Washington, DC, USA. doi:10.1109/SFCS.1977.32. https://doi.org/10.1109/SFCS.1977.32. (cited on pages 1 and 3)

Sarkar, S.; Sewell, P.; Alglave, J.; Maranget, L.; and Williams, D., 2011. Understanding POWER multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11 (San Jose, California, USA, 2011), 175–186. ACM, New York, NY, USA. doi:10.1145/1993498.1993520. http://doi.acm.org/10.1145/1993498.1993520. (cited on pages 18, 23, and 26)

Sewell, P., 2016. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. [Online; accessed 05/11/2018] https://web.archive.org/web/20181016220256/https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. (cited on page 30)

Sewell, P.; Sarkar, S.; Owens, S.; Nardelli, F. Z.; and Myreen, M. O., 2010. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53, 7 (Jul. 2010), 89–97. doi:10.1145/1785414.1785443. http://doi.acm.org/10.1145/1785414.1785443. (cited on pages 15, 16, and 17)

Sorin, D. J.; Hill, M. D.; and Wood, D. A., 2011. *A primer on memory consistency and cache coherence*. No. 16 in Synthesis lectures on computer architecture. Morgan & Claypool, San Rafael, Calif. ISBN 978-1-60845-564-5 978-1-60845-565-2. OCLC: 930741576. (cited on page 15)

Treiber, R. K., 1986. *Systems Programming: Coping with Parallelism*. No. 5118 in Research Report RJ. International Business Machines Incorporated, Almaden Research Center. (cited on page 9)

Wickerson, J.; Batty, M.; Sorensen, T.; and Constantinides, G. A., 2017. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017 (Paris, France, 2017), 190–204. ACM, New York, NY, USA. doi:10.1145/3009837.3009838. http://doi.acm.org/10.1145/3009837.3009838. (cited on page 34)

Zhang, S.; Vijayaraghavan, M.; and Arvind, 2017. Weak memory models: Balancing definitional simplicity and implementation flexibility. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 288–302. doi:10.1109/PACT.2017.29. (cited on pages 15, 18, and 26)