IT 10 068
Examensarbete 30 hp
December 2010

Thread-Modular Model Checking of Concurrent Programs Under TSO Using Code Rewriting

Carl Leonardsson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Thread-Modular Model Checking of Concurrent Programs Under TSO Using Code Rewriting

Carl Leonardsson

Model checking is a well understood method for verifying correctness of concurrent programs. Commonly instructions in the verified programs are assumed to be executed in the order they occur in the programs. Most actual architectures, however, perform a number of optimizations when executing a program. Some such optimizations have the effect of reordering instructions or making the instructions appear to be reordered. The possible code reorderings are governed by the architecture's memory model. The particular memory model considered in this text is called TSO and is used on for example Intel x86 and SPARC V9.

A method for verifying safety properties of concurrent programs under TSO by model checking is presented. The method works by exploring the set of reachable program states to verify that certain predefined error states can never be reached.

The main optimization allowed by TSO is storing write instructions in an on-processor write buffer and later writing through the value to memory asynchronously with the other program instructions. Rather than concretely modelling the write buffers of a TSO architecture, the state exploration described in this paper works by successively rewriting the code of the program being analysed. The rewriting closely corresponds to the instruction reordering analogy sometimes used when describing the semantics of memory models.

An extension of the method is also given, which allows verification of safety properties for programs with an unbounded number of threads.

Handledare: Bengt Jonsson
Ämnesgranskare: Parosh Abdulla
Examinator: Anders Jansson
IT 10068
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

The difficulty of writing bug-free parallel software is well known. To find bugs introduced by the subtle synchronisation issues that can arise from interleaving of instructions, or to assert the absence of such bugs, tools such as SPIN [8] have been developed. Adding to the complexity introduced by instructions from different threads interleaving with each other, instructions from the same thread can even be reordered with each other as a result of optimizations. Such optimizations can be performed by the compiler [4], or by the hardware in the form of instruction level parallelism [12].

While these reorderings are such that they can not be observed by the thread itself they may be observable by other threads when accessing shared memory. Which reorderings may occur in a certain architecture is determined by the *memory model* of the architecture. In the cases where reorderings may be observed by other threads the memory model is called a *relaxed memory model*.

The method described in this text can be used for verification of programs running under relaxed memory models. We will here focus on the specific memory model TSO (Total Store Order) which is used in some common processor families such as SPARC-V9 [16] and Intel x86 [14].

The aim of this method is verifying *safety properties* of concurrent programs. This means verifying that some particular, user-specified, bad state can never be reached when running the program. Examples of bad states could be states where two threads are both in a critical section at the same time or states where data structures break their invariants.

The counterpart of *safety properties* would be *liveness properties*, i.e., the property of a program that some particular, user-specified, good state will eventually be reached when running the program. Verification of liveness properties is currently not supported by the method presented in this document.

In this method we model TSO by reordering the instructions in the program code as described in Section 5. This design choice is meant to keep the model abstract and not concerned with the actual architecture implementation which gives rise to the reordering behavior.

The method requires each thread to be finite state and that the number of shared memory locations is bounded. Further the write-buffer for each thread must be bounded (see Section 5.8). Using the thread-modular extension described in Section 7 the method can, however, be used to verify safety properties for programs with an unbounded number of threads.

In the next few sections, we will start by providing a background describing relaxed memory models focusing on TSO, giving an overview of model checking and discussing related work. In Section 5, we describe how programs under TSO are modelled by code rewriting. The method is presented as a number of rules. In Section 6 a criterion is given which states that the rules of Section 5 correctly model TSO executions of programs. The proof of the criterion is left out due to its size. In Section 7 we describe how model checking can be extended to allow verification of safety properties for an unbounded number of threads. Finally some applications of using the method are presented in Section 8.

In the following text we use the words *thread* and *process* interchangingly. We assume that different threads or processes run on different processors.

2 Relaxed Memory Models and TSO

In this section the concept of *relaxed memory models* (also known as *weak memory models*) will be explained and the particular case of TSO will be covered in more detail.

Typically when writing or reasoning about a program, one expects the instructions to be executed in the order they appear in the source code - to be executed in *program order*. However, when executed on most architectures the instructions can be reordered by optimisation. The optimizations considered here are hardware optimizations. Such optimisations include instruction pipelining, introduction of store buffers in the processor, coalescing of writes to the same cache line and allowing different memory modules to serve different memory accessing instructions without keeping the program order [4].

These optimisations can serve to hide the latency of slow instructions such as memory reads or writes. Common for the optimisations are that they are restricted such that they are impossible to observe for a non-parallel program.

The optimizations can be complex and execute instructions out of order, overlapping and non-atomically. Analogously to [4] we will here use an abstraction where the effects of the optimizations are regarded as reorderings of instructions which are then executed one after another, atomically.

Since an instruction may take some time to be executed, determining an order between instructions requires that we define the precise time when the instruction is deemed to *take effect*. The intuition here should be that the order of two instructions is the order in which they take effect with respect to the global memory.

More precisely,

Definition 1. A writing instruction takes effect at the point in time when the written value *val* reaches the memory.

I.e., a write instruction takes effect at the time after which any read from the same memory location performed by any process will return the value *val*. This point in time will occur some time after the writing instruction is first handled by the executing processor, typically when the value reaches the L1 cache. A processor which supports latency hiding for writing instructions may continue to execute subsequent instructions before the writing instruction has had time to take effect. The subsequent instructions may even take effect before the writing instruction takes effect. In such cases the subsequent instructions and the writing instruction are effectively reordered.

Definition 2. A reading instruction takes effect at the time when the value is fetched from memory.

This may occur some time after the reading instruction is first handled by the executing processor and some time before the value reaches the executing processor and the target register. If the architecture has more than one memory module serving the executing processor it may happen that while the reading instruction is still waiting for memory module A to fetch the requested value at memory location m, a subsequent instruction is issued by the processor which makes a request to memory module B and is served before the value at memory location m is fetched by A; the instructions are effectively reordered.

```
L1: write: x := 1
L2: write: y := 2
```

Figure 1: Instructions that can be reordered transparently to a sequential program.

```
L3: write: x := 1
L4: write: x := 2
```

Figure 2: Instructions that can not be reordered transparently to a sequential program.

By *memory model* or *memory consistency model* we mean the rules governing which reorderings may be performed by optimisations. The memory model where all memory accessing instructions take effect in program order is called *sequential consistency* [10]. Other memory models are called *relaxed memory models* or *weak memory models*.

Example 1. The two instructions `L1` and `L2` in Figure 1 write the value 1 (respectively 2) to the variable x (respectively y) in memory. Whether or not these two instructions are reordered such that the 2 reaches `y` before the 1 reaches `x` can not be observed by a sequential program; whether or not the instructions are reordered the sequential program will see the same values when subsequently reading from variables `x` and `y`.

In Figure 2 the instructions `L3` and `L4` both write to the same variable. Reordering `L3` and `L4` in this case would affect the value returned by a subsequent read from the variable `x` and thus be visible to a sequential program. Therefore the instructions `L3` and `L4` will not be reordered.

Consider now the reordering of instructions `L1` and `L2` in Figure 1 in the context of a parallel program with shared memory. The reordering, while invisible to the process executing the writing instructions, can be observed by another process as we shall see. Let the variables `x` and `y` both have the initial value 0. Let a process P_0 execute the program in Figure 1 and another process P_1 in parallel execute the program shown in Figure 3. If no reordering is allowed then for P_1 after executing both reads it will hold that if `r0` $= 2$ then `r1` $= 1$, since whenever `L5` is executed after `L2` it must hold that `L6` is executed after `L1`. If, on the other hand, the instructions `L1` and `L2` are executed out of order it is possible that `y` is assigned the value 2 by P_0, then P_1 executes both instructions `L5` and `L6` thus assigning `r0` $= 2$ and `r1` $= 0$.

The effects of relaxed memory models in multi-process programs can cause synchronisation to fail. (We will soon see a practical example.) For that reason it is sometimes necessary to enforce program order between particular instruc-

```
L5: read: r0 := y
L6: read: r1 := x
```

Figure 3: A program observing the reordering of Figure 1.

	$W \rightarrow R$ *Order*	$W \rightarrow W$ *Order*	$R \rightarrow R/W$ *Order*	Read others' write early	Read own write early
SC					√
IBM 370	√				
TSO	√				√
PC	√			√	√
PSO	√	√			√
WO	√	√	√		√
RMO	√	√	√		√
PowerPC	√	√	√	√	√

Figure 4: Reorderings allowed by various memory models [4].

tions. One way for the programmer to enforce that order is to use *memory barriers* (also called *memory fences*). Barriers are instructions provided by the architecture which ensure that all memory accessing instructions that have been issued, or a specified subset thereof, have taken effect before any further instructions are issued. See for example the `MEMBAR` instruction of SPARC-V9 [16] or `SFENCE`, `LFENCE` and `MFENCE` of Intel x86 [1].

A *data-race* is defined by Adve and Gharachorloo [4] as follows:

"*Given a sequentially consistent execution, an operation forms a race with another operation if the two operations access the same location, at least one of the operations is a write, and there are no other intervening operations between the two operations under consideration.*"

For relaxed memory models it is common to provide a *DRF (Data-Race Free) guarantee* stating that for programs that cannot contain any data-race when run under sequential consistency, every execution of the program under the relaxed model will correspond to some execution of the program under sequential consistency. Here *correspond* means that every memory reading instruction will read the value written by the same writing instruction in both executions [11] [14]. So, a program using locks to avoid data-races will not be affected by the relaxedness of the memory model. Still, there are applications where locks are undesirable or unavailable such as in libraries where performance is critical or in the implementation of the locks themselves.

Figure 4 shows a number of memory models and which reorderings are allowed under those models.

A memory model allowing the $W \rightarrow R$ relaxation allows the architecture to let a writing instruction be reordered with a subsequent reading instruction, provided the instructions concern different memory locations. This means that if an instruction writing the value val_0 to the variable m_0 is followed by an instruction reading the value of variable m_1 where $m_0 \neq m_1$ then the value of variable m_1 may be fetched from memory before the value of m_0 is updated to val_0 in the memory.

A memory model allowing the $W \rightarrow W$ relaxation allows the architecture to reorder a writing instruction with a subsequent writing instruction provided that the instructions concern different memory locations. This means that if an instruction writing the value val_0 to the variable m_0 is followed by an instruction writing the value val_1 to the variable m_1 where $m_0 \neq m_1$ then it is acceptable

that in the memory the value at location m_1 is updated to val_1 before the value at location m_0 is updated to val_0.

A memory model allowing the $R \to R/W$ relaxation allows the architecture to reorder a reading instruction with a subsequent instruction being either a read or a write, provided the instructions concern different memory locations.

A memory model allowing the *Read others' write early* relaxation allows a process A to read the value written to a memory location m by a process B before that value becomes visible to a third process C. I.e., the following scenario is acceptable. Process B issues an instruction writing the value val to memory location m. Processes A and C both read the value of memory location m at the same time, but for process A the read returns the value val while for process C the read returns the previous value of m. This relaxation is not readily modelled as an instruction reordering and is not further covered in this paper.

A memory model allowing the *read own write early* relaxation allows a process A to read the value previously written to a memory location m by A itself before that value becomes visible to another process B. The instruction writing value val to location m is said to take effect at the time when val reaches the memory. This is also the time when the value val becomes visible to process B. The instruction of process A, reading from location m is said to take effect when the value is fetched. In the case of *read own write early* the value val is fetched by A before it becomes visible to process B. Hence the reading instruction takes effect before the writing instruction takes effect and the *read own write early* can be seen as a reordering of the reading instruction and the writing instruction. This relaxation is further explained in Section 2.1 and in Example 3.

The above rules of reordering are concerned only with types of instructions and which memory locations they access. It should be noted that additional restrictions may be imposed on the reordering of instructions depending on the registers used by the instructions. For example, an instruction that depends on the value of a register r cannot overtake the instruction assigning a value to r.

2.1 Total Store Order (TSO)

The main processor optimisation enabled by TSO is the introduction of write buffers. Writes to memory are time consuming instructions. Therefore when executing a program and encountering a write instruction we do not want to wait for the write to finish before continuing with the execution of subsequent instructions. For that reason, when encountering a write, the target memory location and the value to be written are pushed into an on-chip write buffer. Execution of subsequent instructions continue immediately and the write is written through to cache and memory asynchronously.

Example 2. The code fragments in Figure 5 show one way of implementing a part of Dekker's locking algorithm. When executed under sequential consistency we are guaranteed that in no execution will it hold that `r0 = r1 = 0` when process P_0 has executed `L2` and process P_1 has executed `L4`. In the locking algorithm this corresponds to the impossibility for both processes to enter the critical section.

Let us now introduce write buffers when executing the code of Figure 5.

Process P_0

```
Global variables:
  int flag[0] = 0
  int flag[1] = 0
Registers: r0

L1: write: flag[0] := 1
L2: read: r0 := flag[1]
```

Process P_1

```
Global variables:
  int flag[0] = 0
  int flag[1] = 0
Registers: r1

L3: write: flag[1] := 1
L4: read: r1 := flag[0]
```

Figure 5: Code fragment from Dekker's locking algorithm

Process P_0

```
Global variables:
  int flag[0] = 0
  int flag[1] = 0
Registers: r0

L1: write: flag[0] := 1
L2: barrier
L3: read: r0 := flag[1]
```

Process P_1

```
Global variables:
  int flag[0] = 0
  int flag[1] = 0
Registers: r1

L4: write: flag[1] := 1
L5: barrier
L6: read: r1 := flag[0]
```

Figure 6: Code fragment from Dekker's locking algorithm with barriers

Let process P_0 execute `L1` and push the write into its write buffer. Process P_0 continues to read the value of the variable `flag[1]` from memory into register `r0`. The variable `flag[1]` has at this time a value of 0. Let now process P_1 execute `L3` and push the write of 1 to the variable `flag[1]` into its write buffer. Process P_1 continues to execute line `L4` and reads the value of variable `flag[0]` from memory. Note that the write performed earlier by process P_0 is still in the write buffer on the processor of P_0 and has not yet reached the memory. Hence the value returned to process P_1 by the read from `flag[0]` is 0. Let the executions terminate by finally flushing the write buffers of processes P_0 and P_1 writing the value 1 to variables `flag[0]` and `flag[1]`. We have now reached the forbidden state where `r0 = r1 = 0`. Thus in this implementation of Dekker's algorithm mutual exclusion can not be guaranteed when executed under TSO!

Expressed in terms of reordering, in this execution line `L2` (respectively line `L3`) is reordered with line `L1` (respectively line `L4`) in the sense that they take effect in an order opposite to that of their program order. This is an example of the $W \rightarrow R$ reordering.

To be able to guarantee mutual exclusion in Dekker's algorithm under TSO we must insert memory fences between lines `L1` and `L2` and between lines `L3` and `L4` as seen in Figure 6. A barrier for process P_n will, when executed, block further execution until the write buffer of process P_n is flushed. Thus the reading instruction can not take effect before the writing instruction takes effect; reordering is disabled.

The other relaxation allowed by TSO, as shown in Figure 4, is *read own write*

Process P_0

```
Global variables:
  int x = 0
  int y = 0
Registers: r0 r1

L1: write: x := 1
L2: read: r0 := x
L3: read: r1 := y
```

Process P_1

```
Global variables:
  int x = 0
  int y = 0
Registers: r2 r3

L4: write: y := 1
L5: read: r2 := y
L6: read: r3 := x
```

Figure 7: A program affected by the read own write early relaxation [13].

early. As shown in Example 2 a read may overtake a write provided they access different variables. The read own write early relaxation also allows reordering of a write with a subsequent read which access the *same* variable. What happens is that when executing the reading instruction, rather than waiting for the written value to reach memory before reading it, the value is fetched directly from the write buffer of the processor executing the read.

Example 3. Figure 7 shows a program where the result of the read own write early relaxation is visible [13]. Similar to the code in Figure 5, when executed in sequential consistency the state `r1 = r3 = 0` is impossible when both processes have terminated. Furthermore, in this example that state is impossible if executed under a memory model allowing the $W \rightarrow R$ relaxation but not the read own write early relaxation. The reason is that line `L3` for process P_0 must take effect not until after line `L2` since the $R \rightarrow R$ relaxation is not allowed. Line `L2`, in turn, must take effect not until after line `L1` since the two instructions access the same variable `x`. Symmetrically for process P_1, line `L6` must be executed after line `L4`.

Let us now re-execute the program in Figure 7 but this time allow the read own write early relaxation. First process P_0 executes line `L1` and pushes the write on its write buffer. Process P_0 then executes line `L2`, but rather than reading the value of `x` from memory it fetches the value 1 from its write buffer and assigns `r0 := 1`. Process P_0 continues to execute line `L3`, reads `y = 0` from memory and assigns `r1 := 0`. Process P_1 can now perform a similar execution before the write performed by process P_0 is written through to memory and thus it is possible for processes P_0 and P_1 to terminate in the state `r1 = r3 = 0`.

3 Model Checking

The method presented in this paper is based on reachability analysis using *model checking* [8].

The idea of reachability analysis for a program using model checking is to find the states which are reachable in the program. A state typically means the valuations of all global variables and for each process the value of the program counter and valuations for all local variables.

```
Global variables:
  int l = 0
L1: while(not(CAS(l,0,1))) (* Do nothing *);
L2: (* Critical section *)
L3: l := 0
```

Figure 8: Finite state spin-lock

```
Global variables:
  int l = 0
Local variables:
  int c = 0
L1: while(not(CAS(l,0,1))){
L2:   c := c + 1
   }
L3: (* Critical section *)
L4: l := 0
```

Figure 9: Infinite state spin-lock

A starting state is defined. This would typically be a state where all processes are at the beginning of their programs and all variables are uninitialised. The starting state is by default in the set of reachable states, henceforth $\mathfrak{R}$.

By allowing a process in a reachable state to execute an instruction according to its program, program counter, and global and local variables one or more new states can be reached. Those are added to $\mathfrak{R}$ as well. This process of successive expansion of $\mathfrak{R}$ is continued until a fixpoint is reached, i.e., until each state that can be added to $\mathfrak{R}$ is already in $\mathfrak{R}$.

When $\mathfrak{R}$ has been calculated in this manner, safety properties can be verified by checking that no state in $\mathfrak{R}$ is a bad state according to the specification of bad states.

A necessary requirement for such reachability analysis to work is that the program is *finite state* i.e., that $\mathfrak{R}$ is finite.

Example 4. Figure 8 shows the implementation of a simple spin-lock.

Here the instruction `CAS(x,y,z)` (Compare And Swap) will read the variable `x` and compare it with the value of `y`. If `x` = `y` then `x` will be assigned the value of `z` and `CAS(x,y,z)` will return `true`. Otherwise `CAS(x,y,z)` will return `false` without changing the value of `x`. The instruction `CAS` is executed atomically.

The program shown in Figure 8 is *finite state* since the set of control states ({`L1`,`L2`,`L3`}) is finite and the only variable, `l`, takes values from a finite domain ($\{0, 1\}$).

Figure 9 is the same spin-lock, extended with an local integer variable `c` counting the number of times the process has unsuccessfully attempted to enter the critical section.

The implementation shown in Figure 9 is infinite state because the variable `c` can grow unboundedly as one process has taken the lock and the other repeatedly, and unsuccessfully, attempts to take it. This means that although the two implementations are equivalent with respect to the mutual exclusion property,

only the first one can be analysed using the reachability analysis described in this section.

4 Related Work

This section gives a short list of related articles, without making any claim of completeness. The first article models programs under TSO by an approach different than ours. The second uses yet another model to reach theoretical results about programs under TSO. The third one describes a method for thread-modular model checking, upon which our method is based.

Fence insertion In [9] Kuperstein et al. describe a method which given a program, a safety property and the implementation of a memory model automatically derives positions in the program where barriers can be inserted to satisfy the safety property. The derived set of barrier positions is minimal in the sense that removing any one of them will enable runs of the program where the safety property is violated. The memory model implementation described in [9] explicitly models the write-buffer.

Unbounded buffer In [5] Atig et al. prove the decidability of the reachability problem for TSO by proving that it is equivalent to the reachability problem for lossy channel machines [3]. The latter proof is based on simulation of programs under TSO by lossy channel machines and vice versa.

Thread-modular model checking In [6] Flanagan and Qadeer describe a thread-modular method for model checking. The method described in Section 7 of this document is based on that method.

5 Modelling TSO by Code Rewriting

In this section a method for reachability analysis of programs under TSO is presented.

5.1 The modelling language

We start by examining the language used to describe the targeted programs. The language, henceforth *Lang*, is a small imperative language. It has been constructed such that each atomic statement corresponds to at most one memory accessing machine code instruction. This serves to disambiguate what is meant by program order for memory accesses. All variables in *Lang* are global and shared among all processes. Variables are stored in memory so accessing them requires accessing memory (with the exception of read own write early). Each process may also use an arbitrary number of registers. Registers are local to each process. The registers of one process can not be read nor written by any other process. Registers are kept on chip, so accessing a register does not require accessing memory. By convention we will use the names `r0`, `r1`, ... for registers, and for variables either letters from the end of the alphabet `x`, `y`, ... or more elaborate names such as `flag` or `turn`. Variables and registers are integer

$Identifier ::= \{a-z, A-Z\}\{a-z, A-Z, 0-9, \_\} *$
$Register ::= Identifier$
$Expression ::=$
    $\mathbb{N}$
 | $Register$
 | `(`$Expression$`)`
 | `not(`$Expression$`)`
 | $Expression$ `=` $Expression$
 | $Expression$ `+` $Expression$
 | $Expression$ `-` $Expression$
 | $Expression$ `>` $Expression$
 | $Expression$ `<` $Expression$
 | $Expression$ `&&` $Expression$
 | $Expression$ `||` $Expression$
$Variable ::=$
    $Identifier$
 | $Identifier$`[`$Expression$`]`
$AtomicStatement ::=$
    $Register$ `:=` $Expression$
 | `read:` $Register$ `:=` $Variable$
 | `write:` $Variable$ `:=` $Register$
 | `CMPXCHG(`$Variable_0$ `=` $Register_1$ `?` $Variable_0$ `:=` $Register_0$ `:` $Register_1$ `:=` $Variable_0$`)`
 | `barrier`
$Statement ::=$
    $AtomicStatement$
 | `if(`$Expression$`)`$Statement$
 | `if(`$Expression$`)`$Statement$ `else` $Statement$
 | `while(`$Expression$`)`$Statement$
 | `{`semi colon separated $Statement$ list`}`

Figure 10: Grammar for the modelling language *Lang*.

valued. A special kind of variables is arrays. Arrays are indexed by integers and have no index bounds. The elements of arrays are integer variables, stored in memory and behaving as ordinary variables. There are no functions in *Lang*.

A grammar for *Lang* is given in Figure 10. Expressions (*Expression*) are evaluated to integer values similarly to how they are evaluated in the C language. For the boolean integer expressions (`=`, `<`, `>`, `&&` and `||`) the C convention of representing $false$ by 0 and *true* by all other integer values is used. Note that *Expression* can not contain variables, in contrast perhaps to intuition. The reason for this is to enforce the principle of at most one memory accessing instruction per *Statement* mentioned in the previous paragraph. Thus to use the value from a variable in an expression the value must first be read into a register.

Variables ($Variable$) can be either ordinary variables or arrays indexed by integer expressions.

Atomic statements (*AtomicStatement*) are statements that are executed atomically with respect to reordering and interleaving. The atomic statements are: register assignment, memory read from a variable to a register, memory write from a register to a variable, `CMPXCHG` and memory barrier.

The instruction `CMPXCHG` is a compare-and-exchange instruction from the Intel x86 instruction set [1]. When `CMPXCHG(` v `=` r_1 `?` v `:=` r_0 `:` r_1 `:=` v `)` is executed, the variable v is read from memory and its value compared to that of the register r_1. If the values are equal, the value of register r_0 is written to variable v. Otherwise the value of v is stored in register r_1. Thus to check for success of the instruction, the program would typically compare the value of register r_1 to its previous value to see whether it has changed or not. Note: Don't be fooled by the seemingly flexible syntax $Variable_0$ `=` $Register_1$ `?` $Variable_0$ `:=` $Register_0$ `:` $Register_1$ `:=` $Variable_0$. The variable $Variable_0$ and registers $Register_0$ and $Register_1$ are really just a variable and two registers; they can not be replaced by more general expressions.

The memory barrier enforces that all instructions before the barrier in program order take effect before all instructions that come after the barrier in program order.

Statements (*Statement*) are either atomic statements or compound statements. The latter are: if statements, while statements and blocks.

A block `{`s_0`;` s_1`;`$\cdots s_n$`}` evaluates statements $s_0, s_1 \cdots s_n$ in order from left to right. An empty block `{}` evaluate as a no-op.

An if statement `if(`e`)`s_0[`else`s_1] evaluates as s_0 if e evaluates to a non-zero value. If e evaluates to zero the if statement evaluates as s_1 when the else clause is present and as a no-op otherwise.

A while statement `while(`e`)`s evaluates as a no-op if e evaluates to zero and as `{`s`; while(`e`)`s`}` otherwise.

5.2 Basic concept

$$INIT_{\overline{\mathfrak{R}(g_0,ls_0)}} \quad STEP^{\frac{\mathfrak{R}(g,ls) \quad (g,ls,t)\to(g',l')}{\mathfrak{R}(g',ls[t\leftarrow l'])}}$$

Figure 11: Generalised rules for explicit-state, multi-process reachability analysis.

Figure 11 shows rules that can be used to calculate the set $\mathfrak{R}$ of reachable states for a given number of threads (cf. [6]). Here (g, ls) represents a state where g is a mapping from global variables to their values and ls is a sequence containing a local state per thread. The local state $ls[t]$ is the local state of thread t. The sequence $ls[t \leftarrow l]$ is the sequence ls where the local state of thread t has been replaced by l. The state (g_0, ls_0) is the starting state of the multi process program. The predicate $\mathfrak{R}(g, ls)$ is syntax abuse for $(g, ls) \in \mathfrak{R}$. The predicate $(g, ls, t) \to (g', l')$ is determined by the program code and means that the thread t in the global state g and the local state $ls[t]$ can execute a single atomic instruction such that g is updated to g' and $ls[t]$ is updated to l'.

When analysing under sequential consistency a natural choice of representation for local states would be $ControlState \times LocalStore$ where $ControlState$ is an integer program counter and $LocalStore$ maps registers to their values. But when instructions can be reordered, it is not clear what is meant by an integer program counter. If we chose to let our program counter be equal to the physical program counter on the processor executing our program, we can not for a given program counter value be certain whether or not the instructions that

```
Global variables:
  flag[0] = 0
  flag[1] = 0
```

Process P_0.

```
{
  r0 := 1;
  write: flag[0] := r0;
  read: r1 := flag[1]
}
```

Process P_1.

```
{
  r0 := 1;
  write: flag[1] := r0;
  read: r1 := flag[0]
}
```

Figure 12: The same fragment of Dekker's algorithm as in Figure 5, re-expressed in *Lang*.

have been passed by the program counter have completely taken effect. E.g., in the program in Figure 5, if the program counter is `L2` we can not know whether the write instruction at line `L1` has reached the memory or if it is still waiting in the write buffer. If we instead define the program counter as the position of the last instruction that has taken effect, we will get a similar problem.

The approach used in this method is therefore to represent $ControlState$, not as in integer position, but as *code which has yet to be executed or has yet to take effect*.

Example 5. Let us revisit the code given in Figure 5 and informally show how the rules of Figure 11 together with our definition of control states can be used to reach the same forbidden state as in Example 2. The code is restated in the language *Lang* in Figure 12. Define the starting state as follows:

$$(g_0, ls_0) = ([(\texttt{flag[0]} \mapsto 0), (\texttt{flag[1]} \mapsto 0)], [l_0^0, l_0^1])$$

Here l_0^0 and l_0^1 are the starting local states of processes P_0 and P_1 respectively. They are defined as follows.

$$l_0^0 = (\{\ \texttt{r0 := 1; write: flag[0] := r0; read: r1 := flag[1]}\ \}, [(\texttt{r0} \mapsto 0), (\texttt{r1} \mapsto 0)])$$
$$l_0^1 = (\{\ \texttt{r0 := 1; write: flag[1] := r0; read: r1 := flag[0]}\ \}, [(\texttt{r0} \mapsto 0), (\texttt{r1} \mapsto 0)])$$

This is the starting state, so no instruction has yet been executed. Hence the complete code of processes P_0 and P_1 make up the control states in l_0^0 and l_0^1.

Next we want to extend the set $\mathfrak{R}$ of reachable states by an application of rule $STEP$ in Figure 11. We have in the state (g_0, ls_0) four choices of instructions to let take effect: for process P_0 we can let the assignment `r0 := 1` take effect since it is the first instruction in program order. The writing instruction `write: flag[0] := r0` cannot take effect before the assignment `r0 := 1` since it depends on the value given to `r0`. The reading instruction `read: r1 := flag[1]`, on the other hand, may take effect now since there are no instructions before it that use either register `r1` or variable `flag[1]`. Note here that in TSO we are allowed to let reading instructions overtake writing instructions (Figure 4). The assignment instruction `r0 := 1` does not access memory so the memory model

is not concerned with whether the reading instruction overtakes the assignment. Symmetrically for process P_1 we may let the assignment `r0 := 1` or the read `read: r1 := flag[0]` take effect. Let us pick the assignment instruction for process P_0. We update the local state of P_0 by removing the assignment from its control state and mapping register `r0` to 1. Thus we reach the next state:

$$\begin{array}{l} l_1^0 = \begin{array}{l}(\{\ \texttt{write: flag[0] := r0; read: r1 := flag[1]}\ \},\\ \quad [(\texttt{r0} \mapsto 1), (\texttt{r1} \mapsto 0)])\end{array} \\ l_1^1 = l_0^1 \\ (g_1, ls_1) = (g_0, [l_1^0, l_1^1]) \end{array}$$

In the next step, the instructions that may take effect are for P_1 the same as before. For P_0 we may let the writing instruction `write: flag[0] := r0` take effect since it is now the first instruction in program order and no longer blocked by the assignment to `r0`. For the same reason as before, we may still let the reading instruction `read: r1 := flag[1]` take effect for process P_0. Let us pick the reading instruction, read the value of variable `flag[1]` from the global state and update register `r1` accordingly:

$$\begin{array}{l} l_2^0 = \begin{array}{l}(\{\ \texttt{write: flag[0] := r0}\ \},\\ \quad [(\texttt{r0} \mapsto 1), (\texttt{r1} \mapsto 0)])\end{array} \\ l_2^1 = l_1^1 \\ (g_2, ls_2) = (g_1, [l_2^0, l_2^1]) \end{array}$$

For the next two steps, let process P_1 execute in the same way process P_0 did. First the assignment takes effect.

$$\begin{array}{l} l_3^0 = l_2^0 \\ l_3^1 = \begin{array}{l}(\{\ \texttt{write: flag[1] := r0; read: r1 := flag[0]}\ \},\\ \quad [(\texttt{r0} \mapsto 1), (\texttt{r1} \mapsto 0)])\end{array} \\ (g_3, ls_3) = (g_2, [l_3^0, l_3^1]) \end{array}$$

Then the reading instruction takes effect. Note that no writing instructions have yet taken effect, so the global memory is unchanged and the value returned by the reading instruction will be 0.

$$\begin{array}{l} l_4^0 = l_3^0 \\ l_4^1 = \begin{array}{l}(\{\ \texttt{write: flag[1] := r0;}\ \},\\ \quad [(\texttt{r0} \mapsto 1), (\texttt{r1} \mapsto 0)])\end{array} \\ (g_4, ls_4) = (g_3, [l_4^0, l_4^1]) \end{array}$$

From the state (g_4, ls_4) there are two choices of instruction to take effect. Either `write: flag[0] := r0` for process P_0 or `write: flag[1] := r0` for process P_1. Let us pick the former. The global state is updated such that variable `flag[0]` maps to the value of register `r0` in process P_0.

$$\begin{array}{l} l_5^0 = \begin{array}{l}(\{\},\\ \quad [(\texttt{r0} \mapsto 1), (\texttt{r1} \mapsto 0)])\end{array} \\ l_5^1 = l_4^1 \\ (g_5, ls_5) = ([(\texttt{flag[0]} \mapsto 1), (\texttt{flag[1]} \mapsto 0)], [l_5^0, l_5^1]) \end{array}$$

Allowing the last instruction of process P_1 to take effect in the same way yields the state:

$$
\begin{array}{l}
l_6^0 = \ (\{\texttt{}, \\
\qquad [(\texttt{r0} \mapsto 1), (\texttt{r1} \mapsto 0)]) \\
l_6^1 = \ (\{\texttt{}, \\
\qquad [(\texttt{r0} \mapsto 1), (\texttt{r1} \mapsto 0)]) \\
(g_6, ls_6) = ([(\texttt{flag[0]} \mapsto 1), (\texttt{flag[1]} \mapsto 1)], [l_6^0, l_6^1])
\end{array}
$$

We have reached a point where for both processes all instructions have taken effect and register `r0` has the value 0 for both processes. This is the forbidden state which was also described in Example 2.

5.3 Candidate selection

In the previous section we got an intuition about how instruction reordering can be modelled by selecting instructions out of order from the piece of code representing a control state. In this section and the following that intuition will be made more concrete.

Henceforth given a control state, the instructions that may be the first to take effect in that state will be called *candidate instructions*.

We define a relation $candidates_{TSO} \subseteq (LocalState \times \mathcal{P}(Statement \times Statement))$[1] which relates a local state $l = (q, lstore) : Statement \times (Register \to \mathbb{Z})$ to a set of pairs $(c, q') : Statement \times Statement$ such that c is a candidate of q and q' is q with c removed.

In the typical case, the candidate c will be an instruction picked from q and returned unchanged. In the case of *Read own write early* however, the candidate will be changed. Assume an instruction `read:`r`:=`v is, by read own write early, allowed to overtake an instruction `write:`v`:=`r'. When returned as a candidate the reading instruction will be rewritten to a simple register assignment r`:=`w where w is the value held by r' at the time of the write. This is necessary to model the fact that the value fetched by the read is fetched from the write buffer rather than from memory.

To calculate the candidates of a local state $(q, lstore)$, we traverse the code of q in the same order as a program order execution would. While doing so we keep track of four properties of the code we have passed: the set of registers and variables that have been used, the known valuations of registers and variables, the number of candidates picked so far and whether or not a writing or reading instruction or a memory barrier has been passed. These properties are represented as four entities, together called the *search state*:

- $U \subseteq Variable \bigcup Register$ is the set of used variables and registers.
- $\sigma : (Variable \bigcup Register) \to \mathbb{Z} \bigcup \{\bot\}$ is the valuation of registers and variables. Informally, when searching for candidates for process P it holds that $\sigma(x) = v$ where v is the value of the register or variable x that would be seen if all hitherto passed instructions were executed in program order without any interference from other processes. If it is impossible to

[1] Here, by $\mathcal{P}(S)$ we mean the power set of S.

evaluate v using only the passed instructions and the original valuation of the registers of process P - i.e., without performing a memory read - then $\sigma(x) = \bot$.

- $cnt : \mathbb{N}$ is the number of candidates picked so far.
- $F \subseteq \{write, read, barrier\}$ contains $write$, $read$ and $barrier$ if a writing or reading instruction or a barrier has been passed respectively.

The code traversal is described by the relation $(s, U, \sigma, cnt, F) \xrightarrow{TSO} (c, U', \sigma', cnt', F')$ where s is the original code, U, σ, cnt and F are as described above, c is the set of candidates from s given that U, σ, cnt and F properly describes the code passed before s. The entities U', σ', cnt' and F' have the same interpretation as U, σ, cnt and F but hold after s has been passed.

Given such a relation $\xrightarrow{TSO}$ we can define $candidates_{TSO}$ as follows:

$$\frac{U = \{\} \quad \sigma = \lambda x. \begin{cases} lstore(x) & \text{if } x \in Register \\ \bot & \text{if } x \in Variable \end{cases} \quad cnt = 0 \quad F = \{\} \qquad (q, U, \sigma, cnt, F) \xrightarrow{TSO} (c, U', \sigma', cnt', F')}{candidates_{TSO}((q, lstore), c)} \tag{1}$$

In this section and the next we will define $\xrightarrow{TSO}$ by rules. Doing so we will simplify by considering the elements of c as being only the candidate instructions thus leaving out the code where the candidate has been removed. This is because calculating the code where the candidate has been removed requires quite a lot of book-keeping but is not very difficult. We start with rules for the cases where the code consists of only one atomic statement.

The first rule concerns the case where the statement s is a candidate.

$$\frac{addable(s, U, \sigma, cnt, F)}{(s, U, \sigma, cnt, F) \xrightarrow{TSO} (\{s\}, U[\![s]\!], \sigma[\![s]\!], cnt + 1, F[\![s]\!])} \tag{2}$$

Here the predicate $addable$ defines the requirement for s to be a candidate and $U[\![s]\!]$, $\sigma[\![s]\!]$ and $F[\![s]\!]$ define how the search state is updated when passing s. The predicate $addable$ as well as the search state updates are defined per instruction below.

- Assignment $r :=\! e$:

$$\frac{regs(r :=\! e) \bigcap U = \{\} \quad barrier \notin F}{addable(r :=\! e, U, \sigma, cnt, F)}$$

$$U[\![r :=\! e]\!] = U \bigcup regs(r :=\! e)$$
$$\sigma[\![r :=\! e]\!] = \sigma[r \mapsto \sigma(e)]$$
$$F[\![r :=\! e]\!] = F$$

For an assignment to become a candidate we require that it is not blocked by a barrier preceding it or by another instruction using the same registers.

We update the search state by adding the registers from the assignment to the set of used registers and by updating $\sigma(r)$ to the value of the expression e. Here by $\sigma(e)$ we mean the expression e evaluated when every register r' is valuated as $\sigma(r')$. If e contains a register r' such that $\sigma(r') = \bot$ then $\sigma(e) = \bot$.

- Read: $\texttt{read:}r\texttt{:=}v$:

$$\frac{r \notin U \quad v \notin U \quad \mathit{barrier} \notin F \quad \mathit{read} \notin F}{\mathit{addable}(\texttt{read:}r\texttt{:=}v, U, \sigma, \mathit{cnt}, F)}$$

$$\begin{gathered} U[\![\texttt{read:}r\texttt{:=}v]\!] = U \bigcup \{r, v\} \\ \sigma[\![\texttt{read:}r\texttt{:=}v]\!] = \sigma[r \mapsto \bot] \\ F[\![\texttt{read:}r\texttt{:=}v]\!] = F \bigcup \{\mathit{read}\} \end{gathered}$$

The requirement for a reading instruction to be a candidate is similar to that for an assignment, but here we also require that the variable v has not been used and that this reading instruction does not overtake any preceding reading instruction (cf. Figure 4).

We update the search state by assigning $\sigma(r) = \bot$. This means that when examining other instructions later in the code we can not know the value of r without first executing the memory read $\texttt{read:}r\texttt{:=}v$. Note that when examining the later instructions the reading instruction $\texttt{read:}r\texttt{:=}v$ has been picked as a candidate but not actually been executed. To remember that we have passed a reading instruction we also add *read* to the set F.

- Write: $\texttt{write:}v\texttt{:=}r$:

$$\frac{r \notin U \quad v \notin U \quad \mathit{barrier} \notin F \quad \mathit{read} \notin F \quad \mathit{write} \notin F}{\mathit{addable}(\texttt{write:}v\texttt{:=}r, U, \sigma, \mathit{cnt}, F)}$$

$$\begin{gathered} U[\![\texttt{write:}v\texttt{:=}r]\!] = U \bigcup \{r, v\} \\ \sigma[\![\texttt{write:}v\texttt{:=}r]\!] = \sigma[v \mapsto \sigma(r)] \\ F[\![\texttt{write:}v\texttt{:=}r]\!] = F \bigcup \{\mathit{write}\} \end{gathered}$$

A writing instruction can neither overtake a barrier, a read nor a write so unless $F = \{\}$ a writing instruction can not be picked as a candidate.

We update the search state such that $\sigma(v)$ evaluates to the value of the register r. The meaning of this is the following: Normally knowing the value of the variable v would require us to first perform a memory read. This is represented by $\sigma(v) = \bot$ when the candidate search starts (see equation 1). But instructions following the instruction $\texttt{write:}v\texttt{:=}r$, can instead of reading from memory read the value of r from the write buffer. This is exploited by early reads as can be seen below.

- CMPXCHG: $\texttt{CMPXCHG(}v = r_1\texttt{?}v\texttt{:=}r_0\texttt{:}r_1\texttt{:=}v\texttt{)}$:

$$\frac{r_0 \notin U \quad r_1 \notin U \quad v \notin U \quad \mathit{barrier} \notin F \quad \mathit{read} \notin F \quad \mathit{write} \notin F}{\mathit{addable}(\texttt{CMPXCHG(}v = r_1\texttt{?}v\texttt{:=}r_0\texttt{:}r_1\texttt{:=}v\texttt{)}, U, \sigma, \mathit{cnt}, F)}$$

$$\begin{gathered} U[\![\texttt{CMPXCHG(}v = r_1\texttt{?}v\texttt{:=}r_0\texttt{:}r_1\texttt{:=}v\texttt{)}]\!] = U \bigcup \{r_0, r_1, v\} \\ \sigma[\![\texttt{CMPXCHG(}v = r_1\texttt{?}v\texttt{:=}r_0\texttt{:}r_1\texttt{:=}v\texttt{)}]\!] = \sigma[r_1 \mapsto \bot][v \mapsto \bot] \\ F[\![\texttt{CMPXCHG(}v = r_1\texttt{?}v\texttt{:=}r_0\texttt{:}r_1\texttt{:=}v\texttt{)}]\!] = F \bigcup \{\mathit{read}, \mathit{write}\} \end{gathered}$$

A compare-and-exchange instruction is both a read and a write, therefore it is as limited as a write (which is strictly more limited than a read) when it comes to requirements for being a candidate.

When updating the search state we assign both $\sigma(r_1) = \perp$ and $\sigma(v) = \perp$. This is because, without reading the value of the variable v from memory we can not know whether it is the variable v which has been assigned the value of r_0 or if it is the register r_1 which has been assigned the value of v. Note here that it is not possible for a compare-and-exchange to read the value of v from the write buffer since $write \notin F$ and thus the write buffer must be empty.

- Barrier: `barrier`:

$$\frac{cnt = 0 \quad barrier \notin F}{addable(\texttt{barrier}, U, \sigma, cnt, F)}$$

$$\begin{gathered} U[\![\texttt{barrier}]\!] = U \\ \sigma[\![\texttt{barrier}]\!] = \sigma \\ F[\![\texttt{barrier}]\!] = F \bigcup \{barrier\} \end{gathered}$$

The requirement for a barrier to be a candidate is that it is the first instruction in program order or equivalently $cnt = 0$. This requirement ensures that the barrier does not overtake any instruction.[2]

When updating the search state we add *barrier* to F thus stopping all further instructions from being picked as candidates.

Next we consider the case of read own write early - read from the write-buffer.

$$\frac{addable_{ROWE}(\texttt{read:}r\texttt{:=}v, U, \sigma, cnt, F)}{\begin{gathered}(\texttt{read:}r\texttt{:=}v, U, \sigma, cnt, F) \\ \xrightarrow{TSO} \\ (\{r\texttt{:=}\sigma(v)\}, U[\![\texttt{read:}r\texttt{:=}v]\!], \sigma[\![\texttt{read:}r\texttt{:=}v]\!], cnt + 1, F[\![\texttt{read:}r\texttt{:=}v]\!])\end{gathered}} \tag{3}$$

Here we define the requirement for a read own write early to be performed as follows:

$$\frac{r \notin U \quad \sigma(v) \neq \perp}{addable_{ROWE}(\texttt{read:}r\texttt{:=}v, U, \sigma, cnt, \{write\})}$$

The requirement for a read own write early to be a candidate is the same as that for an ordinary read, but here we additionally require that a write has been performed to the variable v.

Note that when returning the read own write early as a candidate we change it from a read instruction `read:`r`:=`v to an assignment $r\texttt{:=}\sigma(v)$. The value of $\sigma(v)$ is the value written to variable v by a previous write and that was stored in σ when the candidate search processed that write (see $\sigma[\![\texttt{write:}v\texttt{:=}r]\!]$ above). The reason for changing the read instruction to an assignment when returning it as a candidate is that the candidate is supposed to be executed before the corresponding write. Thus keeping the candidate as a read would read the old value of v from memory instead of reading the value placed in the write buffer by the writing instruction (see Example 6).

[2]The extra requirement $barrier \notin F$ is redundant and was added only in order to simplify the proof.

The next rule for $\xrightarrow{TSO}$ concerns the instructions that are not candidates.

$$\frac{atomic(s) \quad \neg addable(s,U,\sigma,cnt,F) \quad \neg addable_{ROWE}(s,U,\sigma,cnt,F)}{(s,U,\sigma,cnt,F) \xrightarrow{TSO} (\{\},U[\![s]\!],\sigma[\![s]\!],cnt,F[\![s]\!])} \tag{4}$$

Here the candidate set is empty and the candidate counter *cnt* is left unchanged.

The next two rules handles blocks of code and should be rather intuitive.

$$(\{\!\}, U, \sigma, cnt, F) \xrightarrow{TSO} (\{\}, U, \sigma, cnt, F) \tag{5}$$

$$\frac{(s,U,\sigma,cnt,F) \xrightarrow{TSO} (c,U'',\sigma'',cnt'',F'') \quad (\{s'\},U'',\sigma'',cnt'',F'') \xrightarrow{TSO} (c',U',\sigma',cnt',F')}{(\{s \cdot s'\},U,\sigma,cnt,F) \xrightarrow{TSO} (c \bigcup c',U',\sigma',cnt',F')} \tag{6}$$

By $x \cdot xs$ we mean x consed onto the list xs.

Example 6. Let us try the rules defined above on an example. In Figure 13 the code from Figure 7 has been rewritten in the language *Lang*. Let us try to find the candidates for process P_0 in the initial state.

We start by applying the rule in Equation (1) which tells us that the candidate set c should satisfy

$(\{\texttt{r4 := 1; write: x := r4; read: r0 := x; read: r1 := y}\},\{\},\sigma,0,\{\}) \xrightarrow{TSO} (c,U',\sigma',cnt',F')$

for some values of U', σ', cnt' and F' where

$$\sigma = \lambda x. \begin{cases} [(\texttt{r0} \mapsto 0),(\texttt{r1} \mapsto 0),(\texttt{r4} \mapsto 0)](x) & \text{if } x \in Register \\ \bot & \text{if } x \in Variable \end{cases}$$

The only $\xrightarrow{TSO}$ rule which is applicable at this point is the one in Equation (6). Applying it tells us that we should first find the candidates for the instruction `r4 := 1`: find c_1, U_1, σ_1, cnt_1 and F_1 such that
$(\texttt{r4 := 1},\{\},\sigma,0,\{\}) \xrightarrow{TSO} (c_1,U_1,\sigma_1,cnt_1,F_1)$
Here depending on the value of $addable(\texttt{r4 := 1},\{\},\sigma,0,\{\})$ we may apply the rule from equation 2 or that from equation 4. In this case $addable(\texttt{r4 := 1},\{\},\sigma,0,\{\})$ holds and `r4 := 1` is indeed a candidate. Equation 2 gives that

$c_1 = \{\texttt{r4 := 1}\}$
$U_1 = U[\![\texttt{r4 := 1}]\!] = U \bigcup regs(\texttt{r4 := 1}) = \{\texttt{r4}\}$
$$\sigma_1 = \sigma[\![\texttt{r4 := 1}]\!] = \lambda x. \begin{cases} [(\texttt{r0} \mapsto 0),(\texttt{r1} \mapsto 0),(\texttt{r4} \mapsto 1)](x) & \text{if } x \in Register \\ \bot & \text{if } x \in Variable \end{cases}$$
$cnt_1 = 1$
$F_1 = F[\![\texttt{r4 := 1}]\!] = F = \{\}$

Equation 6 tells us that when finished with `r4 := 1` we should continue to finding valuations for c c' U', σ', cnt', F' such that

$(\{\texttt{write: x := r4; read: r0 := x; read: r1 := y}\},U_1,\sigma_1,cnt_1,F_1) \xrightarrow{TSO} (c',U',\sigma',cnt',F')$

where $c = c_1 \bigcup c'$.

Another three applications of equation 6 and finally one application of equation 5 splits up the code block into individual instructions and tells us to find valuations for $c_i, U_i, \sigma_i, cnt_i, F_i$ for $i \in \{2,3,4\}$ such that

$(\texttt{write: x := r4}, U_1, \sigma_1, cnt_1, F_1) \xrightarrow{TSO} (c_2, U_2, \sigma_2, cnt_2, F_2)$

$(\texttt{read: r0 := x}, U_2, \sigma_2, cnt_2, F_2) \xrightarrow{TSO} (c_3, U_3, \sigma_3, cnt_3, F_3)$

$(\texttt{read: r1 := y}, U_3, \sigma_3, cnt_3, F_3) \xrightarrow{TSO} (c_4, U_4, \sigma_4, cnt_4, F_4)$
where $c = c_1 \bigcup c_2 \bigcup c_3 \bigcup c_4$. In turn:

- $(\texttt{write: x := r4}, U_1, \sigma_1, cnt_1, F_1) \xrightarrow{TSO} (c_2, U_2, \sigma_2, cnt_2, F_2)$
  As we did for the assignment, we evaluate $addable(\texttt{write: x := r4}, U_1, \sigma_1, cnt_1, F_1)$. This time we find that `write: x := r4` is not a candidate because the register `r4` is already in U_1. Thus $c_2 = \{\}$. According to equation 4 the search state is still updated:
  $U_2 = U_1[\![\texttt{write: x := r4}]\!] = U_1 \bigcup \{\texttt{x}, \texttt{r4}\} = \{\texttt{x}, \texttt{r4}\}$
  $\sigma_2 = \sigma_1[\![\texttt{write: x := r4}]\!] = \lambda x. \begin{cases} [(\texttt{r0} \mapsto 0), (\texttt{r1} \mapsto 0), (\texttt{r4} \mapsto 1)](x) & \text{if } x \in Register \\ \sigma_1(\texttt{r4}) (\text{i.e., } 1) & \text{if } x = \texttt{x} \\ \bot & \text{otherwise} \end{cases}$
  $cnt_2 = 1$
  $F_2 = F_1[\![\texttt{write: x := r4}]\!] = F_1 \bigcup \{write\} = \{write\}$
  Note how the value of register `r4` is stored in $\sigma(\texttt{x})$. This corresponds to pushing the write onto the write-buffer. Any subsequent instruction that is picked as a candidate may read the value of `x` from the write-buffer instead of reading it from memory.

- $(\texttt{read: r0 := x}, U_2, \sigma_2, cnt_2, F_2) \xrightarrow{TSO} (c_3, U_3, \sigma_3, cnt_3, F_3)$
  Evaluating $addable(\texttt{read: r0 := x}, U_2, \sigma_2, cnt_2, F_2)$ tells us that `read: r0 := x` is not a candidate because the variable `x` is already in U_2. On the other hand, $addable_{ROWE}(\texttt{read: r0 := x}, U_2, \sigma_2, cnt_2, F_2)$ holds: the register `r0` has not been used, $\sigma(\texttt{x}) = 1 \neq \bot$ and $F_2 = \{write\}$. Thus the rules in equation 2 and equation 4 are both disabled, but the read own write early rule in equation 3 can be used.

  $c_3 = \{\texttt{r0 := 1}\}$
  $U_3 = U_2[\![\texttt{read: r0 := x}]\!] = U_2 \bigcup \{\texttt{x}, \texttt{r0}\} = \{\texttt{x}, \texttt{r4}, \texttt{r0}\}$
  $\sigma_3 = \sigma_2[\![\texttt{read: r0 := x}]\!] = \lambda x. \begin{cases} [(\texttt{r0} \mapsto \bot), (\texttt{r1} \mapsto 0), (\texttt{r4} \mapsto 1)](x) & \text{if } x \in Register \\ 1 & \text{if } x = \texttt{x} \\ \bot & \text{otherwise} \end{cases}$
  $cnt_3 = 2$
  $F_3 = F_2[\![\texttt{read: r0 := x}]\!] = F_2 \bigcup \{read\} = \{write, read\}$
  Note here that the read `read: r0 := x` has been replaced as a candidate by the assignment `r0 :=` $\sigma_2(\texttt{x}) =$`1` to capture that the value was fetched from the write-buffer.

- $(\texttt{read: r1 := y}, U_3, \sigma_3, cnt_3, F_3) \xrightarrow{TSO} (c_4, U_4, \sigma_4, cnt_4, F_4)$
  This time neither $addable(\texttt{read: r1 := y}, U_3, \sigma_3, cnt_3, F_3)$ nor $addable_{ROWE}(\texttt{read: r1 := y}, U_3, \sigma_3, cnt_3, F_3)$ holds because $read \in F_3$. Thus `read: r1 := y` is not a candidate and we apply the rule in equation 4 to get

```
Global variables: x = y = 0
```

Process P_0

```
Initially: r0 = r1 = r4 = 0

{
  r4 := 1;
  write: x := r4;
  read: r0 := x;
  read: r1 := y
}
```

Process P_1

```
Initially: r2 = r3 = r5 = 0

{
  r5 := 1;
  write: y := r5;
  read: r2 := y;
  read: r3 := x
}
```

Figure 13: A program affected by the read own write early relaxation [13]. The *Lang* version of the program in Figure 7

$$
\begin{aligned}
&c_4 = \{\} \\
&U_4 = U_3[\![\texttt{read: r1 := y}]\!] = U_3 \bigcup \{\texttt{y}, \texttt{r1}\} = \{\texttt{x}, \texttt{y}, \texttt{r4}, \texttt{r0}, \texttt{r1}\} \\
&\sigma_4 = \sigma_3[\![\texttt{read: r1 := y}]\!] = \lambda x. \begin{cases} [(\texttt{r0} \mapsto \bot), (\texttt{r1} \mapsto \bot), (\texttt{r4} \mapsto 1)](x) & \text{if } x \in \mathit{Register} \\ 1 & \text{if } x = \texttt{x} \\ \bot & \text{otherwise} \end{cases} \\
&cnt_4 = 2 \\
&F_4 = F_3[\![\texttt{read: r1 := y}]\!] = F_3 \bigcup \{\mathit{read}\} = \{\mathit{write}, \mathit{read}\}
\end{aligned}
$$

Finally we can see that $c = c_1 \bigcup c_2 \bigcup c_3 \bigcup c_4 = \{\texttt{r4 := 1}, \texttt{r0 := 1}\}$.

5.4 Loops

In this section we introduce loops and rules allowing $\xrightarrow{TSO}$ to handle them.

Informally, the way loops are handled is that they are unrolled on demand such that candidates can be selected from the code which is no longer inside the loop.

Example 7. Given a control state

```
{
  r0 := 3;
  while(r0){
    read: r1 := x;
    write: y := r1;
    r0 := r0 - 1
  }
}
```

we want to be able to select (among others) a candidate (`read: r1 := x`, q') where `read: r1 := x` is the candidate instruction and q' is the control state where the candidate has been removed:

```
{
```

```
  r0 := 3;
  write: y := r1;
  r0 := r0 - 1;
  while(r0){
    read: r1 := x;
    write: y := r1;
    r0 := r0 - 1
  }
}
```

Loops are handled by $\xrightarrow{TSO}$ by five rules. If-statements are handled similarly and will not be described in this document. To make the method more accessible, let us first consider only the special case where all loops are guaranteed by the programmer to deterministically terminate. Here by *deterministically* we mean without depending on values communicated by the memory or by other processes.

5.4.1 First attempt: no infinite loops

Assume the candidate search has reached a while loop `while(`e`)`s. There may be candidates in the first iteration of the loop, in any later iteration of the loop, and after the loop. Now evaluate $\sigma(e)$ in the current search state. There are three alternatives:

- Assume $\sigma(e) = 0$
  In this case we know that there can be no candidate in any iteration of the loop since the loop will never be executed. Handle the loop as a no-op and return an empty candidate set and an unchanged search state.

- Assume $\sigma(e) \in \mathbb{Z} - \{0\}$
  In this case we know that the first iteration of the loop will be executed. Therefore, recursively select all candidates from s (and append `while(`e`)`s to the end of the new control states which are returned together with the candidates). Then recursively select all candidates from `while(`e`)`s with the new search state produced by s. We know that this recursion will terminate from the guarantee provided by the programmer that all loops will terminate deterministically.

- Assume $\sigma(e) = \perp$
  This means that the value of e depends on a value read from memory. Thus the loop must be preceded by a reading operation. Therefore it holds that $read \in F$. In the rules for *addable* we can see that when $read \in F$, no instruction is a candidate. This corresponds to the fact that the $R \rightarrow R/W$ relaxation is disabled for TSO (see Figure 4). For this reason, there can not be any candidates neither in the loop nor after the loop. In this case we may handle the loop as a no-op; returning an empty set of candidates and an unchanged search state.

  Above we stated that no instruction can be a candidate. As the scrutinising reader might have noticed there is one exception: the assignment instruction which does not access memory. This means that under the rule set described in this document we can not guarantee that *all* candidates

are found for a given control state. Rather we limit ourselves to the weaker claim that *all memory accessing* candidates are picked in every order permitted by TSO. This weaker claim is formalised and partially proven in Section 6. Note that as soon as the blocking read instruction has been executed the blocked assignments may be executed and thereafter possible memory accessing instructions that were blocked by the assignments. Since the reorderings of TSO are such that they can not be observed by the process itself the reorderings can only be observed through communication with other processes. Therefore, to find observable violations of safety conditions, it is enough to maintain the reordering behaviour for the communicating - memory accessing - instructions.

5.4.2 Adding infinite loops

The rules given in the previous section do not work for programs that may contain infinite loops. But there are programs that contain infinite loops or that contain loops that are infinite when run in isolation, which we would like to analyse. For example: The implementation of spin-lock given in Figure 8 contains a loop which is infinite if run in isolation after another process has taken the lock.

We define a set $Terminates$ as follows. The pair $(\texttt{while}(e)s, \sigma)$ is a member of $Terminates$ iff the program $\texttt{while}(e)s$ when executed with register valuations as given by σ will in a finite number of steps either terminate or execute a reading instruction (`read` or `CMPXCHG`) or a barrier. The set $Terminates$ can not be computed. It will be approximated later in this section.

We note that if the candidate search reaches a loop $\texttt{while}(e)s$ in a search state where $(\texttt{while}(e)s, \sigma) \in Terminates$ then we can find all candidates by using the rules defined in the previous section until either the loop terminates or the search encounters a reading instruction or barrier. If the loop terminates then searching may continue as usual. If a reading instruction or a barrier is encountered the search may stop early since there can be no more candidates after the reading instruction or barrier.

Assume alternatively that the candidate search reaches a loop $\texttt{while}(e)s$ in a search state where $(\texttt{while}(e)s, \sigma) \notin Terminates$. In this case, from the definition of $Terminates$ we know that the loop will never execute any memory accessing instruction other than writing instructions. Writing instructions can not be reordered with each other since the $W \rightarrow W$ relaxation is disabled in TSO. We also know that there can be no candidates in the code after the loop since the loop will never terminate. In this case the candidate search will only unroll the loop for as long as it can find candidates in the next iteration of the loop. There can be candidates in only a finite number of loop iterations since there is only a finite number of instructions in the loop body and as soon as an instruction is picked as a candidate it will be blocked by itself in the subsequent iterations. (Assignment instructions block subsequent copies of themselves by using the same target register. Other instructions block themselves by adding elements to the set F in the search state.) Since the only memory accessing instructions are writes and they may not be reordered under TSO, limiting ourselves in this way to only a prefix of the complete run of the loop each time we select candidates does not limit the possible reorderings of memory accessing instructions.

For a general loop `while`$(e)s$ and search state to calculate whether (`while`$(e)s, \sigma) \in Terminates$ is impossible (equivalent to the stop-problem) assuming a Turing-complete *Lang* together with possibly infinite-state programs. Assuming finite-state programs, it is costly.

The method used by our candidate selection algorithm is instead the following. Together with the search state we keep a concrete and finite set $Terminates^*$ of pairs (l, σ) of loops and register stores which are known to be members of $Terminates$. When the candidate search encounters a loop l in a search state (U, σ, cnt, F) it checks if the pair (l, σ) is in $Terminates^*$. If so, then the loop will terminate or execute a reading instruction or barrier and it is safe to handle the loop l as described above.

If (l, σ) is not in $Terminates^*$, then the loop l will be handled as if it were not a member of $Terminates$. While doing so, every time l is unrolled, the value of σ at that point is inserted into a set $queried(l)$ associated with l and with the current branch of the reachability analysis. There are two cases to consider:

If $(l, \sigma) \in Terminates$ for some $\sigma \in queried(l)$ then per definition of $Terminates$, the reachability analysis will sooner or later reach a state where l terminates or executes a reading instruction or barrier. When that is detected the pairs (l, σ) for all $\sigma \in queried(l)$ are inserted into $Terminates^*$. Then for all states in the reachability analysis where a pair (l, σ) has been tested for membership in $Terminates^*$, where (l, σ) is equal to one of the pairs just inserted into $Terminates^*$, the reachability analysis is restarted. Thus for all states in the reachability analysis, where it is necessary to know for candidate selection whether a loop l is in $Terminates$ and where l actually is in $Terminates$, we will sooner or later be able to run the candidate selection algorithm knowing that l is in $Terminates^*$ and thus in $Terminates$.

Let us now consider the case where $(l, \sigma) \notin Terminates$ for all $\sigma \in queried(l)$. Per definition of $Terminates$ we know that the loop l is an infinite one when starting in any of the register valuations in $queried(l)$. Since we also know that the program is finite-state (by assumption) we know that at some time l will revisit a reachability analysis state. That can be detected by the reachability analysis, which can then handle the infinite loop by standard model checking methods (see e.g., [7]). In the meanwhile we know by the discussion above of handling of loops not in $Terminates$ that no TSO reorderings will be lost.

5.4.3 The rules

In the previous section an argument was given for how loops can be handled in candidate selection. In this section the actual rules are given.

$$\frac{\sigma(e) = 0}{(\texttt{while}(e)s, U, \sigma, cnt, F) \xrightarrow{TSO} (\{\}, U, \sigma, cnt, F)} \tag{7}$$

This rule handles the simple case where we know that the loop will not be executed.

$$\frac{\sigma(e) = \bot}{(\texttt{while}(e)s, U, \sigma, cnt, F) \xrightarrow{TSO} (\{\}, U, \sigma, cnt, F \bigcup \{barrier\})} \tag{8}$$

This rule handles the case where the value of e depends on a memory read. There can be no memory accessing candidates in the loop. Instructions that do

not access memory are here ignored as candidates since we would otherwise be forced to speculate on the execution of the loop. The *barrier* is added to F to ensure that no instructions that do not access memory are picked as candidates from the code following the while.

$$\frac{\sigma(e) \in \mathbb{Z} - \{0\} \quad (s, U, \sigma, cnt, F) \xrightarrow{TSO} (\{\}, U', \sigma', cnt', F') \quad (\texttt{while}(e)s, \sigma) \notin Terminates^*}{(\texttt{while}(e)s, U, \sigma, cnt, F) \xrightarrow{TSO} (\{\}, U, \sigma, cnt, F \bigcup \{barrier\})} \tag{9}$$

This rule handles the case where we do not know if the loop is in $Terminates$ and there are no candidates in the next iteration of the loop. The *barrier* is added to F in order to stop subsequent instructions from being picked as candidates. Picking subsequent candidates would be speculating on the termination of the loop. In the cases where the loop actually will terminate and a subsequent candidate could and should be picked, the loop is certain to later in the analysis be inserted into $Terminates^*$ and the analysis restarted from this point.

$$\frac{\sigma(e) \in \mathbb{Z} - \{0\} \quad (s, U, \sigma, cnt, F) \xrightarrow{TSO} (c, U'', \sigma'', cnt'', F'') \quad c \neq \{\} \quad (\texttt{while}(e)s, U'', \sigma'', cnt'', F'') \xrightarrow{TSO} (c', U', \sigma', cnt', F')}{(\texttt{while}(e)s, U, \sigma, cnt, F) \xrightarrow{TSO} (c \bigcup c', U', \sigma', cnt', F')} \tag{10}$$

$$\frac{\sigma(e) \in \mathbb{Z} - \{0\} \quad (s, U, \sigma, cnt, F) \xrightarrow{TSO} (c, U'', \sigma'', cnt'', F'') \quad (\texttt{while}(e)s, \sigma) \in Terminates^* \quad (\texttt{while}(e)s, U'', \sigma'', cnt'', F'') \xrightarrow{TSO} (c', U', \sigma', cnt', F')}{(\texttt{while}(e)s, U, \sigma, cnt, F) \xrightarrow{TSO} (c \bigcup c', U', \sigma', cnt', F')} \tag{11}$$

The two rules above handle the complement case to the one handled in equation 9: Either the loop is known to be in $Terminates$ or there are candidates in the next iteration of the loop. In both cases we unroll.

$$\frac{\{read, barrier\} \bigcap F \neq \{\}}{(s, U, \sigma, cnt, F) \xrightarrow{TSO} (\{\}, U, \sigma, cnt, F \bigcup \{barrier\})} \tag{12}$$

This rule allows candidate selection to terminate as soon as a reading instruction or a barrier has been passed. This is safe because there can be no memory accessing candidates after a reading instruction or a barrier has been passed. The reason *barrier* is added to F is to make sure that no register assigning instructions after this point are selected as candidates. If they were they might illegally overtake instructions in s.

5.5 Arrays

In the language *Lang*, array elements can be used wherever variables can. To support the use of arrays the rules for candidate selection must be extended.

The first adaptation is to add the requirement that in order for an instruction which accesses array element $a\texttt{[}e\texttt{]}$ to be a candidate it must hold that the

registers used in e have not been used in any passed instructions: $regs(e) \bigcap U = \{\}$.

Next comes the question whether the memory location itself has been used in any passed instructions. This require us to know for all earlier accesses of array elements $a[e']$ whether $a[e']$ is the same as $a[e]$, i.e., whether e and e' will evaluate to the same index. Here we use an argument similar to that used when reasoning about loop conditions: If either $\sigma(e) = \bot$ or $\sigma(e') = \bot$ then the candidate search must have passed a reading instruction. Therefore the considered instruction can not be a candidate. If on the other hand both $\sigma(e) \neq \bot$ and $\sigma(e) \neq \bot$ we may evaluate the expressions to see whether they index the same position in the array.

5.6 Modelling in *Lang*

There are a few things to consider when implementing algorithms in *Lang*.

Compilers and high-level language The instruction ordering in a program written in *Lang* is supposed to correspond to the instruction ordering in the *machine code* of the modelled program. The reorderings performed by the candidate selection correspond to those performed on the machine code by the processor. A program written in a high-level language may be optimised by the compiler in ways which may affect the possible instruction reorderings. Such optimisations include keeping a variable in a register instead of in memory and code restructuring such as loop unrolling or elimination of common subexpressions. Therefore a *Lang* model built from code written in a high-level language might not be an appropriate model for behaviour of the actual compiled program.

Registers are not local variables Seeing that a state consists of a global state with global variables and a local state with registers it may be tempting to equate registers and local variables. Remember though, that a local variable might be kept in memory and thus accessing it could block subsequent candidate instructions where accessing a register would not. To model a local variable one could instead use a global variable which, by programmer discipline, only one process ever accesses.

Avoid introducing false dependencies. Consider the two programs in Figure 14. Running the candidate selection algorithm on the left one gives us two candidates: `r0 := 1` and `read: r1 := y`. In the one to the right we get only one candidate: `r0 := 1`. In this case the instruction `read: r0 := y` is blocked from being a candidate because the register `r0` has been used before. But let us consider a run of the right program: First execute `r0 := 1`. Then execute the instruction `write: x := r0`, but only insert $(\texttt{x}, 1)$ into the write-buffer. Now execute `read: r0 := y` and finally flush the write-buffer. In this run we were able to allow `read: r0 := y` overtake the write even though they use the same register. The reason is that once the value of `r0` has been read and inserted into the write-buffer the register itself is no longer necessary and the processor may allow `read: r0 := y` to overwrite `r0` with the value of `y`.

```
{                                  {
  r0 := 1;                           r0 := 1;
  write: x := r0;                    write: x := r0;
  read: r1 := y                      read: r0 := y
}                                  }
```

Figure 14: Two ways of implementing a fragment of Dekker's algorithm.

The rules for candidate selection enforce a too strict policy on dependencies caused by register reuse. This is an error in the current algorithm and should be corrected as a part of "further work".

A correction would probably require that the way in which each register has been used is kept track of in more detail by the candidate selection algorithm, rather than only keeping track of which registers have been used. It would probably also require some rewriting of the instructions that are being overtaken. For example in the case of the right hand code in Figure 14, if the read is allowed to overtake the write then the write must be rewritten to `write: x := 1`. Otherwise the read would store tho value of `y` in the register `r0` which would then be written to the variable `x`.

Note, however, that it is possible to work around the problem by not reusing registers in cases where it is not necessary to communicate information stored in them. E.g., in this case use the left implementation rather than the right one.

5.7 Extending this method to other memory models

When the prototype implementing this modelling method was created, apart from TSO also the memory model of IBM 370 was implemented. The method could also be extended to PSO by changing the rules to allow writes to be candidates even if writes to different memory locations have already been passed.

More relaxed memory models such as RMO, which also have the $R \rightarrow R/W$ relaxation could not be modelled by this method without some major and unknown changes. The reason is the way loops are handled. The current way of loop handling avoids speculation by terminating the candidate search when a reading instruction has been passed. If the $R \rightarrow R/W$ relaxation is allowed, such an early termination might miss candidates.

5.8 Limitations of this method

In this section three main limitations to the usability of this method are discussed.

Programs must be finite-state. As mentioned in Section 3 the program of each thread must be finite-state. In some cases it is possible to model an infinite-state program as a finite-state program by abstraction.

For example, in some cases it is possible to replace a variable i ranging over the complete $\mathbb{Z}$ with a small number of boolean-valued variables each of which represents some property of i such as "i is even", "i is greater than k" etc. This method of abstraction was used to represent the arbitrarily large pointer version numbers when analysing Treiber's algorithm (see Section 8.3).

```
{
  r0 := 1;
  while(r0){
    read: r0 := x;
    write: y := r1
  }
}
```

Figure 15: Program where the write-buffer may grow unboundedly.

```
{
  write: y := r1;
  while(r0){
    read: r0 := x;
    write: y := r1
  }
}
```

Figure 16: The program of Figure 15 when two instructions have been executed.

Programs must have bounded write-buffers. The number of elements in the write-buffer can grow unboundedly for certain programs. This can make the programs infinite-state if the buffers are modelled explicitly in the local states. Although this method of modelling TSO does not explicitly model the write-buffer, the corresponding problem occurs even in this method.

Consider the program in Figure 15. Similar code occurs in e.g., Burn's locking algorithm. Suppose that `x` is 1 in memory. The loop body can be executed infinitely many times (or in the case of Burn's, arbitrarily many times until the other processes back off). In each iteration a write $(\mathtt{y}, \sigma(\mathtt{r1}))$ can be added to the write buffer.

In the code rewriting method for modelling TSO the problem manifests itself in the following manner: The first candidate will be `r0 := 1`. After that the loop will be unrolled once. The next candidate is `read: r0 := x`. At this point we have reached the control state (i.e., remaining code) shown in Figure 16. Note that the write which is now the first instruction of the program comes from the first loop body which was unrolled. From this control state let us

```
{
  write: y := r1;
  write: y := r1;
  while(r0){
    read: r0 := x;
    write: y := r1
  }
}
```

Figure 17: The program of Figure 16 after another read has been executed.

unroll the loop once more and select `read: r0 := x` as the next candidate. (Note that the read is a possible candidate since it may overtake the writes.) This puts us in the control state shown in Figure 17. This control state differs from the previous one because there are now *two* writes before the loop. We may continue producing new unique control states indefinitely in this way by always selecting the read for candidate. Thus the program is now infinite-state and the model checking analysis will not terminate.

Intra-instruction dependencies are too strong. See the discussion in Section 5.6.

6 Correctness of candidate selection

The rules for candidate selection which are given in Section 5.3 and 5.4 have been formalised in the theorem prover Isabelle. Independently, using program order semantics and straightforward application of the TSO reordering rules, the association between programs and possible TSO traces were formalised. As an argument for the correctness of the candidate selection rules a (partial) correctness criterion was stated for the rules based on the association of programs and TSO traces. The criterion has been partially proven to hold in Isabelle.

In this section I will state the rules both for candidate selection and TSO traces. I will also give the correctness criterion which was partially proven in Isabelle. The complete Isabelle proof is too large to be included in this report.

6.1 Candidate selection rules

The rules given in this section are almost the same as the ones given in Sections 5.3 and 5.4. They differ in that there is no rule which corresponds the rule given in equation 12. The rules used in the Isabelle definitions should be extended to also include such a rule.

When the rules refer to the set $Terminates$ they refer to the actual set $Terminates$ rather than the approximation $Terminates^*$. The motivation for this is that for each loop it will sooner or later be handled based on the correct assumption of whether or not it is in $Terminates$. At that time we are guaranteed to find the correct set of candidates, but not until then.

In the rule $ReadOwnWriteEarly$ we introduce a new statement type $\texttt{read}_{loc}\texttt{:}r\texttt{:=}e$. This statement is the same as an ordinary register assignment $r\texttt{:=}e$ but with the additional information that it is a read statement that has been transformed by a read own write early. This information allows us to handle the statements correctly when defining the TSO reorderings in the next section.

$$\frac{regs(r\texttt{:=}e) \bigcap U = \{\} \quad barrier \notin F}{addable(r\texttt{:=}e, U, \sigma, cnt F)}$$

$$\frac{r \notin U \quad v \notin U \quad barrier \notin F \quad read \notin F}{addable(\texttt{read:}r\texttt{:=}v, U, \sigma, cnt, F)}$$

$$\frac{r \notin U \quad v \notin U \quad barrier \notin F \quad read \notin F \quad write \notin F}{addable(\texttt{write:}v\texttt{:=}r, U, \sigma, cnt, F)}$$

$$\frac{r0 \notin U \quad r1 \notin U \quad v \notin U \quad barrier \notin F \quad read \notin F \quad write \notin F}{addable(\texttt{CMPXCHG(}r0,r1,v\texttt{)}, U, \sigma, cnt, F)}$$

$$\frac{cnt{=}0 \quad barrier \notin F}{addable(\texttt{barrier}, U, \sigma, cnt, F)}$$

$$\frac{r \notin U \quad \sigma(v) \neq \bot}{addable_{ROWE}(\texttt{read:}r\texttt{:=}v, U, \sigma, cnt, \{write\})}$$

$U[\![r\texttt{:=}e]\!] = U \bigcup regs(r\texttt{:=}e)$	$\sigma[\![r\texttt{:=}e]\!] = \sigma[r \mapsto \sigma(e)]$	$F[\![r\texttt{:=}e]\!] = F$
$U[\![\texttt{read:}r\texttt{:=}v]\!] = U \bigcup \{r, v\}$	$\sigma[\![\texttt{read:}r\texttt{:=}v]\!] = \sigma[r \mapsto \bot]$	$F[\![\texttt{read:}r\texttt{:=}v]\!] = F \bigcup \{read\}$
$U[\![\texttt{write:}v\texttt{:=}r]\!] = U \bigcup \{r, v\}$	$\sigma[\![\texttt{write:}v\texttt{:=}r]\!] = \sigma[v \mapsto \sigma(r)]$	$F[\![\texttt{write:}v\texttt{:=}r]\!] = F \bigcup \{write\}$
$U[\![\texttt{CMPXCHG(}r0, r1, v\texttt{)}]\!] = U \bigcup \{r0, r1, v\}$	$\sigma[\![\texttt{CMPXCHG(}r0, r1, v\texttt{)}]\!] = \sigma[r1 \mapsto \bot][v \mapsto \bot]$	$F[\![\texttt{CMPXCHG(}r0, r1, v\texttt{)}]\!] = F \bigcup \{read, write\}$
$U[\![\texttt{barrier}]\!] = U$	$\sigma[\![\texttt{barrier}]\!] = \sigma$	$F[\![\texttt{barrier}]\!] = F \bigcup \{barrier\}$

$$End \quad (\{\}, U, \sigma, cnt, F) \xrightarrow{TSO} (\{\}, U, \sigma, cnt, F)$$

$$Concat \frac{(s,U,\sigma,cnt,F) \xrightarrow{TSO} (c,U'',\sigma'',cnt'',F'') \quad (\{s'\},U'',\sigma'',cnt'',F'') \xrightarrow{TSO} (c',U',\sigma',cnt',F')}{(\{s \cdot s'\},U,\sigma,cnt,F) \xrightarrow{TSO} (c \bigcup c',U',\sigma',cnt',F')}$$

$$Candidate \frac{addable(s,U,\sigma,cnt,F)}{(s,U,\sigma,cnt,F) \xrightarrow{TSO} (\{s\},U[\![s]\!],\sigma[\![s]\!],cnt+1,F[\![s]\!])}$$

$$ReadOwnWriteEarly \frac{addable_{ROWE}(\texttt{read:}r\texttt{:=}v,U,\sigma,cnt,F)}{(\texttt{read:}r\texttt{:=}v,U,\sigma,cnt,F) \xrightarrow{TSO} (\{\texttt{read}_{loc}\texttt{:}r\texttt{:=}\sigma(v)\},U \bigcup \{r,v\},\sigma[r \mapsto \bot],cnt+1,F \bigcup \{read\})}$$

$$NoCandidate \frac{atomic(s) \quad \neg addable(s,U,\sigma,cnt,F) \quad \neg addable_{ROWE}(s,U,\sigma,cnt,F)}{(s,U,\sigma,cnt,F) \xrightarrow{TSO} (\{\},U[\![s]\!],\sigma[\![s]\!],cnt,F[\![s]\!])}$$

$$WhlBase \frac{\sigma(e)=0}{(\texttt{while}(e)s,U,\sigma,cnt,F) \xrightarrow{TSO} (\{\},U,\sigma,cnt,F)}$$

$$WhlIneval \frac{\sigma(e)=\bot}{(\texttt{while}(e)s,U,\sigma,cnt,F) \xrightarrow{TSO} (\{\},U,\sigma,cnt,F \bigcup \{barrier\})}$$

$$WhlNoExp \frac{\sigma(e) \in \mathbb{Z}-\{0\} \quad (s,U,\sigma,cnt,F) \xrightarrow{TSO} (\{\},U',\sigma',cnt',F') \quad (\texttt{while}(e)s,\sigma) \notin Terminates}{(\texttt{while}(e)s,U,\sigma,cnt,F) \xrightarrow{TSO} (\{\},U,\sigma,cnt,F \bigcup \{barrier\})}$$

$$WhlExp0 \frac{\sigma(e) \in \mathbb{Z}-\{0\} \quad (s,U,\sigma,cnt,F) \xrightarrow{TSO} (c,U'',\sigma'',cnt'',F'') \quad c \neq \{\} \quad (\texttt{while}(e)s,U'',\sigma'',cnt'',F'') \xrightarrow{TSO} (c',U',\sigma',cnt',F')}{(\texttt{while}(e)s,U,\sigma,cnt,F) \xrightarrow{TSO} (c \bigcup c',U',\sigma',cnt',F')}$$

$$WhlExp1 \frac{\sigma(e) \in \mathbb{Z}-\{0\} \quad (s,U,\sigma,cnt,F) \xrightarrow{TSO} (c,U'',\sigma'',cnt'',F'') \quad (\texttt{while}(e)s,\sigma) \in Terminates \quad (\texttt{while}(e)s,U'',\sigma'',cnt'',F'') \xrightarrow{TSO} (c',U',\sigma',cnt',F')}{(\texttt{while}(e)s,U,\sigma,cnt,F) \xrightarrow{TSO} (c \bigcup c',U',\sigma',cnt',F')}$$

$$Start \frac{U=\{\} \quad \sigma=\{(r,v) | r \in Register \wedge v=value(r)\} \bigcup Variable \times \{\bot\} \quad cnt=0 \quad F=\{\} \quad (s,U,\sigma,cnt,F) \xrightarrow{TSO} (c,U',\sigma',cnt',F')}{c_{TSO}(s,c)}$$

Above $value(r)$ refers to the current value of register r in the local state of the process at the time that the candidate selection algorithm is started. We also use $\texttt{CMPXCHG}(r0, r1, v)$ as a short form for $\texttt{CMPXCHG}(v = r1\texttt{?}v\texttt{:=}r0\texttt{:}r1\texttt{:=}v)$.

6.2 Rules for TSO traces

The rules for $trace_{seq}$ define the program order traces of programs written in the language *Lang*. The rules for $reorder_{TSO}$ define TSO traces based on sequential traces.

To know which traces are possible we must know the evaluation of the expression e every time we encounter a loop $\texttt{while}(e)s$. Here we solve the problem by making a trace for each of the possible branches (corresponding to $\sigma(e) = 0$ and $\sigma(e) \neq 0$). In each of the cases we insert a $\texttt{require}(e')$ instruction where $e' = e$ or $e' = \texttt{not}(e)$. The interpretation is that a trace is impossible if $\sigma(e') = 0$ at the point in the code of the $\texttt{require}(e')$ statement. The relevant trace for a certain starting σ and for certain interaction with other processes is the one where all $\texttt{require}(e')$ are such that $\sigma(e') \neq 0$ when the statement is executed.

Rules for if-statements have been left out since they are a simpler case of while-statements.

$$\frac{atomic(s) \quad i \in \mathbb{ID}}{trace_{seq}(s, <s_i>)}$$

$$trace_{seq}(\{\}, <>)$$

$$\frac{trace_{seq}(s,t) \quad trace_{seq}(s',t') \quad ids(t) \bigcap ids(t') = \{\}}{trace_{seq}(\{s \cdot s'\}, t @ t')}$$

$$\frac{i \in \mathbb{ID}}{trace_{seq}(\texttt{while(}e\texttt{)}s, <\texttt{require(not(}e\texttt{))}_i>)}$$

$$\frac{trace_{seq}(s,t) \quad trace_{seq}(\texttt{while(}e\texttt{)}s,t') \quad i \in \mathbb{ID} \quad i \notin ids(t) \bigcup ids(t') \quad ids(t) \bigcap ids(t') = \{\}}{trace_{seq}(\texttt{while(}e\texttt{)}s, \texttt{require(}e\texttt{)}_i \cdot t @ t')}$$

$$local(r\texttt{:=}e_{id}) \quad local(\texttt{require(}e\texttt{)}_{id})$$

$$TsoEq \quad reorder_{TSO}(t,t)$$

$$TsoAnyLoc \frac{local(b) \quad regs(a) \bigcap regs(b) = \{\} \quad a \neq \texttt{barrier}_{id} \quad reorder_{TSO}(s, t @ <a,b> @ t')}{reorder_{TSO}(s, t @ <b,a> @ t')}$$

$$TsoLocAny \frac{local(b) \quad regs(a) \bigcap regs(b) = \{\} \quad a \neq \texttt{barrier}_{id} \quad reorder_{TSO}(s, t @ <b,a> @ t')}{reorder_{TSO}(s, t @ <a,b> @ t')}$$

$$TsoWriteRead0 \frac{r \neq r' \quad v \neq v' \quad reorder_{TSO}(s, t @ <\texttt{write:}v'\texttt{:=}r'_i, \texttt{read:}r\texttt{:=}v_j> @ t')}{reorder_{TSO}(s, t @ <\texttt{read:}r\texttt{:=}v_j, \texttt{write:}v'\texttt{:=}r'_i> @ t')}$$

$$TsoWriteRead1 \frac{r \neq r' \quad reorder_{TSO}(s, t @ <\texttt{write:}v\texttt{:=}r'_i, \texttt{read:}r\texttt{:=}v_j> @ t')}{reorder_{TSO}(s, t @ <\texttt{read}_{loc}\texttt{:}r\texttt{:=}r'_j, \texttt{write:}v\texttt{:=}r'_i> @ t')}$$

$$TsoSeq \frac{trace_{seq}(s,t) \quad reorder_{TSO}(t,t')}{trace_{TSO}(s,t')}$$

6.3 Additional definitions

We start by defining predicates $\mathbb{T}[\cdot]$ to simplify speaking of different types of statements.
$\mathbb{T}[\texttt{:=}](a) \equiv \exists r, e.a = r\texttt{:=}e$
$\mathbb{T}[\texttt{require}](a) \equiv \exists e.a = \texttt{require(}e\texttt{)}$
$\mathbb{T}[\texttt{read:}](a) \equiv \exists r, v.a = \texttt{read:}r\texttt{:=}v$
$\mathbb{T}[\texttt{read}_{loc}\texttt{:}](a) \equiv \exists r, r'.a = \texttt{read}_{loc}\texttt{:}r\texttt{:=}r'$
$\mathbb{T}[\texttt{write:}](a) \equiv \exists r, v.a = \texttt{write:}v\texttt{:=}r$
$\mathbb{T}[\texttt{CMPXCHG}](a) \equiv \exists r_0, r_1, v.a = \texttt{CMPXCHG(}r_0, r_1, v\texttt{)}$
$\mathbb{T}[\texttt{barrier}](a) \equiv a = \texttt{barrier}$

$\mathbb{T}\{t_0, t_1, ...\}(a) \equiv \exists n \in \mathbb{N}.\mathbb{T}[t_n](a)$

We define a function $update(\sigma, t)$ which describes how the state σ is updated by the execution of the trace t:

$$update(\sigma,t)=\begin{cases}\sigma & \text{if } t=<>\\ update(\sigma[\![s]\!],t') & \text{if } t=s\cdot t'\wedge\mathbb{T}\{\texttt{:=},\texttt{read:},\texttt{write:},\texttt{CMPXCHG},\texttt{barrier}\}(s)\\ update(\sigma,t) & \text{if } t=s\cdot t'\wedge\mathbb{T}\{\texttt{require}\}(s)\\ update(\sigma[r\mapsto\sigma(e)],t) & \text{if } t=\texttt{read}_{loc}\texttt{:}r\texttt{:=}e\cdot t'\end{cases}$$

We define a predicate $impossible(t,\sigma)$ with the interpretation that the trace t is certain to be "impossible" (as described in Section 6.2) when starting in a state as described by σ. We note that all traces for programs that do not terminate are *impossible* since all traces are finite. Here $_{@}$ means list concatenation.

$$impossible(t,\sigma)\equiv\exists u,e,u'.t=u_{@}<\texttt{require(}e\texttt{)}>{}_{@}u'\wedge update(\sigma,u)(e)=0$$

Next we define predicates $P_{(i)}(j)$ with the interpretation that it is possible for an instruction i to overtake an instruction j.

$$\begin{array}{ll}P_{r\texttt{:=}e}(j)\equiv & \neg\mathbb{T}[\texttt{barrier}](j)\wedge regs(j)\bigcap regs(r\texttt{:=}e)=\{\}\\ P_{\texttt{read:}r\texttt{:=}v}(j)\equiv & \neg\mathbb{T}\{\texttt{barrier},\texttt{read:},\texttt{read}_{loc}\texttt{:},\texttt{CMPXCHG}\}(j)\wedge r\notin regs(j)\\ P_{\texttt{read}_{loc}\texttt{:}r\texttt{:=}r'}(j)\equiv & \neg\mathbb{T}\{\texttt{barrier},\texttt{read:},\texttt{read}_{loc}\texttt{:},\texttt{CMPXCHG}\}(j)\wedge r\notin regs(j)\wedge\\ & r'\notin regs(j)\\ P_{\texttt{write:}v\texttt{:=}r}(j)\equiv & \neg\mathbb{T}\{\texttt{barrier},\texttt{read:},\texttt{read}_{loc}\texttt{:},\texttt{write:},\texttt{CMPXCHG}\}(j)\wedge r\notin\\ & regs(j)\wedge v\notin vars(j)\\ P_{\texttt{CMPXCHG(}r_0,r_1,v\texttt{)}}(j)\equiv & local(j)\wedge r_0\notin regs(j)\wedge r_1\notin regs(j)\\ P_{\texttt{barrier}}(j)\equiv & false\end{array}$$

Based on that definition we define a predicate $\mathfrak{C}_{TSO}(t,i)$ which means that the instruction with id i in the trace t can overtake all instructions before it (with the possible exception of `require`).

$$\mathfrak{C}_{TSO}(t,i)\equiv t=u_{@}<a_i>{}_{@}u'\wedge\forall b\in u.\mathbb{T}[\texttt{require}](b)\vee P_{(a)}(b)$$

Next we define the set $Terminates$ which occurs in the rules for $\xrightarrow{TSO}$. $Terminates$ contains precisely the pairs $(\texttt{while(}e\texttt{)}s,\sigma)$ such that $\texttt{while(}e\texttt{)}s$ when executed from the starting state σ, sooner or later, will either terminate or execute a `read:`, `CMPXCHG` or `barrier`.

$$Terminates=\left\{(\texttt{while(}e\texttt{)}s,\sigma)\;\middle|\;\begin{array}{l}\exists t,a,u,u'.\\ \quad trace_{seq}(\texttt{while(}e\texttt{)}s,t)\wedge\\ \quad\left(\begin{array}{l}\neg impossible(t,\sigma)\vee\\ \left(\begin{array}{l}t=u_{@}<a>{}_{@}u'\wedge\\ \neg impossible(u,\sigma)\wedge\\ \mathbb{T}\{\texttt{read:},\texttt{barrier},\texttt{CMPXCHG}\}(a)\end{array}\right)\end{array}\right)\end{array}\right\}$$

6.4 Correctness criterion

$$\begin{array}{l}trace_{tso}(s,t)\wedge\\ (\forall x.\sigma(x)=\bot\iff x\in Variables)\wedge\\ \neg impossible(t,\sigma)\wedge\\ first\_instr(t,a)\wedge \qquad (a \text{ is the first non-}\texttt{require}\text{ statement in } t.)\\ \neg local(a)\wedge\\ (s,\{\},\sigma,0,\{\})\xrightarrow{TSO}(c,U',\sigma',cnt',F')\\ \Rightarrow\\ \exists b.b\in c\wedge(a=b\vee(\exists r,e,e'.a=\texttt{read}_{loc}\texttt{:}r\texttt{:=}e\wedge b=\texttt{read}_{loc}\texttt{:}r\texttt{:=}e'))\end{array}$$

What we state is that if an instruction a is the first one in the trace t for a program s, then a is picked by the candidate selection algorithm.

If a would happen to be an instruction $\texttt{read}_{loc}\texttt{:}r\texttt{:=}e$ then we only require from the candidate selection algorithm that the corresponding candidate is a read own write early instruction to the same register. I.e., we allow the expressions e and e' to differ. Note also that this correctness criterion is limited to finite traces since we require that $\neg impossible(t, \sigma)$. Both of the above-stated are unnecessary weaknesses to the correctness criterion and should be eliminated.

7 Thread-Modularity

In order to verify safety properties for programs with an arbitrary number of threads rather than for only a particular number of threads we introduce a method for thread modular model checking. It is an extension of the method described by Flanagan and Qadeer in [6].

The method used in this work will first be described, then differences to the method used by Flanagan and Qadeer will be discussed.

The goal is calculating the set $\mathfrak{R}$ of reachable states. Let G be the set of global states; the set of mappings from variables to values. Let L take the corresponding role for local states; be the set of mappings from registers to values. Let P be the set of control states, i.e., remaining code. Further define the *class* of a thread t as the set of all threads with the same initial control state as t. Let C be a set of identifiers associated with thread classes. We can now represent a thread at a given point of its execution as an element of the set $C \times P \times L$. Let L_C be the set of multisets of threads: $L_C = (C \times P \times L) \to \mathbb{N}$. A system state with a global variable store and a number of different threads can be represented as an element of the set $S = G \times L_C$. Those are the kind of states contained in $\mathfrak{R} \subseteq S$.

We define the program semantics by assuming a set $\mathfrak{T} \subseteq C \times P \times L \times G \times P \times L \times G$ such that $(c, p, l, g, p', l', g') \in \mathfrak{T}$ means that a thread (c, p, l) in the global state g, can execute an instruction such that the thread transitions into (c, p', l') while changing g to g'. In the method described in this document, membership in $\mathfrak{T}$ is calculated on demand by candidate selection from the control state and execution of the selected candidate.

Now the thread-modular idea is to pick a small positive number N (typically 1 or 2) and limit the reachability analysis to states (g, l_c) where the number $|l_c|$ of concretely represented threads equals N. Now change the meaning of the statement $(g, l_c) \in \mathfrak{R}$ to mean that there exists a set of threads l_{cloud} such that the state $(g, l_c \bigcup l_{cloud})$ is reachable from the starting state where all threads are at their starting positions. The intuition here is that the thread set l_{cloud} is the set of all threads in the larger state that are left implicit in the concretely represented set l_c. To simulate the behaviour of the implicit threads, we capture the changes to the global state performed by the concretely represented threads in l_c. We then allow an implicit thread t_i not in l_c to apply the same changes to the state (g, l_c) whenever certain requirements for t_i and the state (g, l_c) are satisfied. Those requirements are that g should be the same as it was when the concrete thread first performed the changes. We also require that replacing any concrete thread in l_c with the implicit thread t_i gives a state which is reachable.

Let $\mathfrak{G} \subseteq C \times P \times L \times G \times G$ be the set of global transitions performed by concrete threads. Here the global transition $(c, p, l, g, g') \in \mathfrak{G}$ is a transition

performed by a thread (c, p, l) in a global state g where the transition changed g into g'.

We may now formulate the rules for thread-modular reachability analysis:

$$Init\ \frac{|l_c|=N \quad \forall (c,p,l)\in l_c.[\quad (p,l)=(I_P(c),I_L(c)) \quad C_\sharp(l_c,c)\leq_{Many} I_N(c)]}{\mathfrak{R}(I_G,l_c)}$$

$$Step\ \frac{\mathfrak{R}(g,l_c) \quad (c,p,l)\in l_c \quad \mathfrak{T}(c,p,l,g,p',l',g') \quad l'_c=l_c[(c,p',l')/(c,p,l)]}{\mathfrak{R}(g',l'_c) \quad \mathfrak{G}(c,p,l,g,g')}$$

$$Env\ \frac{\mathfrak{R}(g,l_c) \quad \mathfrak{G}(c,p,l,g,g') \quad C_\sharp(l_c,c)+1\leq_{Many} I_N(c) \quad \forall t\in l_c.\mathfrak{R}(g,l_c[(c,p,l)/t])}{\mathfrak{R}(g',l'_c)}$$

Here $I_P(c)$, $I_L(c)$, $I_N(c)$ and I_G are respectively the starting control state for threads of class c, the starting local state for threads of class c, the maximal number of threads of class c in any state and the initial global state. The number of threads is a natural number $\leq N$ or the special value $Many$ which means that any number is allowed. The expression $C_\#(l_c, c) \leq_{Many} n$ means that either $n = Many$ or the number of threads of class c in l_c is lesser or equal to the natural number n.

The first rule thus initialises $\mathfrak{R}$ with all states containing precisely N concrete threads where all threads are in their initial states and the global state is I_G.

The second rule specifies that in case a concrete thread in a reachable state can execute an instruction then the state reached by the execution is also reachable and additionally, the corresponding transition is in the set $\mathfrak{G}$. Here $l_c[a/b]$ means l_c where one element b has been replaced by one element a.

The last rule specifies that an implicit thread (c, p, l) may perform a transition from $\mathfrak{G}$ in case the global part g of the current state (g, l_c) matches the global state of the transition. We also require that at some point in the analysis it has been shown that a state is reachable which has the global part g and has concrete threads like the ones in l_c except that one concrete thread has been replaced by the implicit thread (c, p, l). This last requirement ensures for example that in a state where one concrete thread is in a critical section we do not allow an implicit thread which is also in the critical section to perform a global transition.

Comparison with Flanagan's and Qadeer's method [6] The method described by Flanagan and Qadeer in [6] is similar to the one described above for the special case of $N = 1$. In the same paper they describe the analysis of a simple locking algorithm. They describe a problem that occurs because implicit

threads can perform global transitions corresponding to instructions within the critical section even when the lock is taken by the concrete thread. The reason is that the information of which process has taken the lock is invisible in the global state and the prerequisites of an implicit thread performing a global transition depend only on the global state. Flanagan and Qadeer solve the problem by including the thread identifier of the locking process in the lock in the global state.

The method described in this section solves the same problem without making any changes to the analysed program but by adding the prerequisite that the implicit thread must be able to coexist with the concrete threads.

8 Results

The method described in this document was implemented as a prototype. The most noteworthy of the tests performed were tests of Dekker's locking algorithm, Peterson's locking algorithm and Treiber's lock-free stack. All tests were run on an Intel i3 2.26 GHz on a single thread.

8.1 Dekker's algorithm

As mentioned in Section 2.1 Dekker's algorithm require memory barriers to guarantee mutual exclusion when run under TSO.

The implementation (called Dekker in the table below) used in this test was taken from the SPARC-V9 architecture manual [16] and contains barriers meant for TSO. It passed verification without breaking mutual exclusion.

Next the implementation was modified by removing in turn each of the two[3] barriers (called Dekker' and Dekker" in the table below). Verification found states where mutual exclusion was broken for both implementations.

Algorithm	Property	Errors	Barriers	Threads	States	Time
Dekker	Mutex		2	2	403	0.056s
Dekker'	Mutex	√	1	2	1028	0.112s
Dekker"	Mutex	√	1	2	676	0.080s

8.2 Peterson's algorithm

The results for Peterson's algorithm are similar to those for Dekker's. In the table below the second row is Peterson's algorithm without barriers. It does not guarantee mutual exclusion under TSO. The first row is the same algorithm with one barrier inserted such that mutual exclusion is guaranteed.

[3]The original implementation contained one barrier, but when rewritten in *Lang* a loop had to be unrolled once due to *Lang* not supporting loops where the loop-condition is first checked *after* the first iteration.

Algorithm	Property	Errors	Barriers	Threads	States	Time
Peterson	Mutex		1	2	366	0.028s
Peterson'	Mutex	$\surd$	0	2	1226	0.136s

8.3 Treiber's algorithm

In [15] a lock-free algorithm for pushing and popping elements of a LIFO stack is described. It is given, in that paper, first in a form where it is vulnerable to the ABA problem, then in a correct form where the ABA problem has been eliminated by the introduction of a version counter for the pointer to the head of the stack.

The implementation in *Lang* includes a simple memory management system since *Lang* has no native support for memory allocation or deallocation. The memory was bounded such that the stack could contain at most two elements at any time.

The analysis was searching for bad states where the stack contains loops or points to memory locations which have not been allocated.

In the table below, the first row represents the faulty implementation. The ABA problem was detected by the reachability analysis.

The third row represents the correct algorithm as suggested by Treiber in [15] where a version counter is used by the code for element popping only.

The second row represents an implementation where both the pushing and the popping code uses the version counter. For both the second and third implementations, the reachability analysis did not find any bad states.

The analysis was performed thread-modularly with 2 concrete threads.

Treiber's stack algorithm does not require any barriers to work under TSO. This is because the synchronising instructions are `CMPXCHG`, which itself acts as a barrier.

Algorithm	Property	Errors	Barriers	Threads	States	Time
Treiber	Data Inv.	$\surd$	0	(∞, ∞)	144105	7min 31s
Treiber'	Data Inv.		0	(∞, ∞)	138881	4min 33s
Treiber"	Data Inv.		0	(∞, ∞)	155816	4min 10s

9 Conclusions and Further Work

The method described in this paper can be used to verify safety properties for programs of an unbounded number of individually finite state threads sharing memory. The method requires that the number of memory locations used is bounded and that the write buffer of each thread is bounded. It handles both finite and infinite loops automatically. The method avoids details of the architecture implementation by modelling the effects of hardware optimizations as reordering of the program source code.

Some possibilities for further work have already been mentioned above. Other possibilities are the following.

The method could be adapted to work for PSO. Adding support for more relaxed memory models like RMO would require a completely new way of handling e.g., loops and arrays, since the current solutions are based on the $R \rightarrow R/W$ relaxation being disabled.

The analysis could also be augmented with a more abstract state representation. By introduction of some abstraction for the values of integer variables, e.g., gap-order constraints, it would be possible to support variable values ranging over the complete set of natural numbers rather than over a small subset thereof. It would also be possible to run the analysis once for an infinite set of starting states rather than having to run once per concrete mapping from variables to starting values.

Another direction of abstraction would be to introduce some more general representation of memory locations. One such representation is the heap signatures given in [2]. With such a representation it should be possible to for example verify Treiber's algorithm with unbounded heap size.

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 1: Basic Architecture*. 2010.

[2] Parosh Aziz Abdulla, Muhsin Atto, Jonathan Cederberg, and Ran Ji. Automatic verification of dynamic data-dependent programs, 2009. http://user.it.uu.se/~parosh/publications/papers/atva09.pdf.

[3] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *Proc. LICS' 93* 8^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 160–170, 1993.

[4] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[5] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *POPL '10 Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2010.

[6] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN Model Checking and Software Verification: Proc.* 10^{th} *Int. SPIN Workshop*, pages 213–224, 2003.

[7] Gerard J. Holzmann. *Spin Model Checker, The Primer and Reference Manual*. Addison Wesley, September 2003.

[8] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.

[9] Michael Kuperstein, Matrin Vechev, and Eran Yahav. Automatic inference of memory fences, 2010.

[10] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28:690–691, 1979.

[11] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05 Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005.

[12] David A. Patterson and John L. Hennessy. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[13] Peter Sewell. Relaxed memory models in 2010. In *UPMARC Summer School on Multicore Computing, June 21-24, 2010*, 2010.

[14] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53, 2010.

[15] R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr., 1986.

[16] David L. Weaver and Tom Germonds, editors. *The SPARC Architecture Manual: Version 9*. PTR Prentice Hall, 1994.