# MEMORAX, a precise and sound tool for automatic fence insertion under TSO ⋆

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Yu-Fang Chen[2], Carl Leonardsson[1], and Ahmed Rezine[3]

[1] Uppsala University, Sweden
[2] Academia Sinica, Taiwan
[3] Linköping University, Sweden

**Abstract.** We introduce MEMORAX, a tool for the verification of control state reachability (i.e., safety properties) of concurrent programs manipulating finite range and integer variables and running on top of weak memory models. The verification task is non-trivial as it involves exploring state spaces of arbitrary or even infinite sizes. Even for programs that only manipulate finite range variables, the sizes of the store buffers could grow unboundedly, and hence the state spaces that need to be explored could be of infinite size. In addition, MEMORAX incorporates an interpolation based CEGAR loop to make possible the verification of control state reachability for concurrent programs involving integer variables. The reachability procedure is used to automatically compute possible memory fence placements that guarantee the unreachability of bad control states under TSO. In fact, for programs only involving finite range variables and running on TSO, the fence insertion functionality is complete, i.e., it will find all minimal sets of memory fence placements (minimal in the sense that removing any fence would result in the reachability of the bad control states). This makes MEMORAX the first freely available, open source, push-button verification and fence insertion tool for programs running under TSO with integer variables.

## 1 Introduction

We introduce MEMORAX, the first freely available, open source (`https://github.com/memorax/memorax`), push-button verification and fence insertion tool that can handle integer variables and that is both sound and complete under TSO for all programs that only involve finite range variables. Modern concurrent processor architectures allow *weak* (*relaxed*) memory models, in which certain memory operations may overtake each other. The use of weak memory models makes reasoning about behaviours of concurrent programs challenging, even for skilled developers. This is for instance witnessed by the lively debate among developers on the Linux Kernel Mailing list about the correctness on x86 of the "Linux Ticket Lock" protocol. (See the mail thread starting with `https://lkml.org/lkml/1999/11/20/76`.) In fact, several synchronisation algorithms, such as mutual exclusion and producer-consumer protocols, turn out to be

⋆ This research was in part funded by the Uppsala Programming for Multicore Architectures Research Center (UPMARC), the National Science Council of Taiwan project no. NSC-101-2221-E-001-007, and the CENIIT research organization (project 12.04).

incorrect if run without modification on weak memories [5]. MEMORAX is based on the techniques developed in [4] and extended in [3]. Not only does our tool turn this verification task into a push-button exercise, it also automatically inserts fences in order to ensure correctness of programs that were made incorrect by the weak memory relaxation. More precisely:

- MEMORAX is an open source [2] push-button tool that comes with a graphical user interface and a simple low level language with a well defined semantics.
- it is sound for concurrent programs running on the TSO memory model (i.e., x86 and SPARC platforms) and involving variables with finite or integer ranges.
- it performs reachability on infinite state spaces to verify control state reachability.
- it provides users with concrete counter-examples, useful for debugging, that take the program from an initial configuration to a specified bad control state.
- it is complete, using an intricate encoding based on the theory of well-quasi-ordering, for the reachability problem of programs on TSO, provided that they only have finite range variables.
- it uses an off-the-shelf SMT solver (MathSAT [1]) to incorporate an interpolation based CEGAR loop to handle integer variables.
- it automatically finds (sets of) fences to ensure a safety property is respected if the property does hold on SC.
- it finds all minimal sets of fences for programs with finite range variables and running on TSO.

**Targeted user base** We see three potential groups of users for MEMORAX :

1. Computer science researchers can use the open source code of MEMORAX to compare with other approaches for the verification of programs running on top of weak memory models, to improve and optimise the implemented techniques (e.g. by interfacing with other SMT solvers or by improving the used data structures or the symbolic representations), or to target new platforms and programs (e.g. add soundness for RMO or PSO, or scale for heap manipulating programs)
2. Teachers of architecture and concurrent programming classes can use (and augment) MEMORAX with its simple user interface in order to familiarise their students with weak memory models. In particular the precision and counter example capabilities of MEMORAX can concretely illustrate the effects of relaxed memory.
3. Software developers working on complex and low level, lock-free code can use MEMORAX to easily check the effects of TSO on their tentative solutions. The generated error traces are also possible on weaker memory models and can conveniently help to highlight possible problems.

**Related tools and approaches** As far as we know, MEMORAX is the first available open source verification and fence insertion tool that is sound on TSO, that can handle integer variables, and that is complete for programs with finite range variables under TSO. There exists several very conservative approaches that restrict to SC executions by establishing "triangular race freedom" [12] or by inserting fences using "delay set

analysis" [13]. We will not further elaborate on those techniques, but will instead focus on a number of tools and approaches more similar to our own.

*CheckFence* [6] is a SAT-based tool that tests correctness of fence placements by considering finite executions on different relaxed memory models. The tool cannot verify programs that result in buffers of arbitrary size like the ones MEMORAX handles since it unrolls loops and checks correctness of the resulting finite executions.

*Fender* [8, 9] combines model checking with abstraction in order to perform reachability analysis on finite over-approximations. It considers different memory models and uses the reachability analysis to justify fence placements. The analysis is not exact and cannot guarantee to show absence of errors for correct programs. As a result, the tool lacks the precision that would allow it to find minimal sets of fence placements. Unfortunately, we were not able to find the tool which is, as far as we know, not open source. Finally, the tool does not handle programs with integer variables.

*mmchecker* [7] performs explicit model-checking for the .NET memory model. It explores the (possibly infinite) state space and inserts fences in order to forbid behaviours that are not possible under SC. The tool cannot prove correctness of programs that generate infinite state spaces but do not require fences. Also the tool cannot soundly handle integer variables like MEMORAX does on TSO.

*Automata based accelerations* [10, 11] computes under-approximations of the generated infinite state space on different relaxed memory models. When the analysis terminates, it answers exactly whether the property is violated or not, and it allows to deduce minimal sets of fence placements, even for programs that may generate buffers of arbitrary sizes. The approach targets systems that manipulate finite variables. It neither can handle integer variables nor does it guarantee termination. We were not able to get hold of the tool or of its source code.

```
1  forbidden
2    CS CS
3
4  data
5    turn = * : [0:1]
6    x    = 0 : [0:1]
7    y    = 0 : [0:1]
```

```
9  process
10 registers
11   $r0 = * : [0:1]
12   $r1 = * : [0:1]
13 text
14   L0: write: x := 1;
15    write: turn := 1;
16   L1: read: $r0 := y;
17   read: $r1 := turn;
18   if $r0 = 1 && $r1 = 1 then
19     goto L1;
20   CS: write: x := 0;
21   goto L0
```

```
23 process
24 registers
25   $r0 = * : [0:1]
26   $r1 = * : [0:1]
27 text
28   L0: write: y := 1;
29   write: turn := 0;
30   L1: read: $r0 := x;
31   read: $r1 := turn;
32   if $r0 = 1 && $r1 = 0 then
33     goto L1;
34   CS: write: y := 0;
35   goto L0
```

**Fig. 1.** Peterson's mutual exclusion protocol.

## 2 Using the tool

### 2.1 The RMM language

Programs to be tested with MEMORAX are written in the special purpose language RMM. For reasoning about programs under relaxed memory, detailed knowledge about

how variables are stored and used is necessary. RMM is designed to unambiguously describe that aspect by making memory accesses and register use explicit.

As an example, Figure 1 shows an RMM model of the Peterson mutual exclusion protocol. Lines 1-2 are of particular interest, since they specify the safety criterion: It is forbidden for the processes (henceforth called `P0` and `P1`) to simultaneously be in the control states labelled `CS` (i.e. line 20 for `P0` and line 34 for `P1`).

### 2.2 Usage through the Graphical Interface

The GUI is a python script (`memorax-gui`) wrapping around the CLI. The GUI window consists of three main parts: The command input area, the code area and the output area. The command input area provides the commands "Reachability", "Fence insertion" and "Draw automata", and options for the commands. All commands apply to the code in the code area, and print their output (and possibly errors) to the output area.

A typical work flow would be the following: First write the RMM code for the protocol you want to analyse. Then use the "Draw automata" command to produce a PDF file showing the automata for the defined processes. This is useful for asserting that the RMM code specifies what you intended. Next use the "Reachability" command to check whether the protocol is safe from the start. If not, then use the "Fence insertion" command to receive sets of fences that will make the protocol safe.

**Reachability** The Reachability command is used to analyse whether there is some configuration which violates the safety specification, but is reachable from some initial configuration. If there is such a configuration, then an error trace will be supplied.

There are currently two reachability methods ("abstractions") available in MEMORAX: SB ("Single Buffer") and PB ("Predicate abstraction and buffer Bounding"), corresponding respectively to our works in [4] and [3]. The PB method is an over-approximation and allows for CEGAR abstraction refinement.

Protocols can be automatically rewritten to "*Register Free Form*" before being analysed. This encodes register values in control states, and can often improve analysis performance.

**Fence insertion** The fence insertion command will repeatedly execute reachability queries, while gradually adding fences to the analysed protocol in order to guarantee satisfaction of the safety criterion. The available options for fence insertion are the same as for reachability, and apply to the repeated reachability queries.

*Interpreting the Output:* If we apply the fence insertion command to the program in Figure 1, we will get output describing the results of the reachability queries. There will be a description of the result at the end of the output:

```
Found 1 fence set:
Fence set #0:
  L15 P0: write: turn := 1
  L29 P1: write: turn := 0
```

Here MEMORAX has found exactly one minimal and sufficient set of fences, namely the one corresponding to locking the writes at line 15 and 29. Other possible outcomes include the empty set - meaning the program is already correct, and no sets - meaning the program cannot be corrected with fences.

## 3 Implementation

MEMORAX is implemented in C++ with the intent of being easy to extend with new memory models and analysis methods.

**Reachability optimisations.** We mention some of the techniques we use to combat the state space explosion problem:

- *Light-Weight Pre-Analysis.* Before the reachability analysis is started, we apply a light-weight, per-thread, over-approximating analysis. This allows us to collect a rough invariant about the buffer contents that are possible per control state and process. We use the invariant to efficiently reduce the explored state space.
- *Update Restriction.* We soundly limit store buffer updating to only take place after a read instruction by the same process. The rationale is that it is only relevant to delay a write instruction by buffering if it is delayed past a read instruction. Other delays can be simulated under SC.
- *Partial Order Reduction for TSO.* In addition to the above update limitation, MEMORAX uses a partial order reduction technique based on the principle that an instruction reordering that does not participate in a conflict cycle, as defined in [13], can be simulated by an appropriate scheduling under SC. Thus instruction reorderings that do not participate in conflict cycles need not be analysed.

**Fence insertion.** The fence insertion algorithm relies on the underlying reachability analysis when evaluating each fence set placement. It is therefore desirable to keep the number of tried fence sets as small as possible.

- *Fence Placement Restriction.* We restrict the number of possibilities, by only considering fences that can be added by locking some write instruction. For example, changing `write: x := 1` into `locked write: x := 1` adds a fence after `write: x := 1`. This guarantees finding minimal and sufficient fence sets (if they exist). Their size can however be larger than a smallest sufficient set.
- *Multiple Fence Extraction.* We perform an extensive analysis to capture fences that need to be added in order to avoid a given error trace. By identifying the conflict cycles (as described by [13]) that a particular reordering $(a \to b)$ participates in, it is sometimes possible to deduce the existence of another, similar error trace where $a$ and $b$ occur in program order, but another pair $(c \to d)$ is reordered, yielding the same conflict cycle. In such cases a fence between $c$ and $d$ is equally necessary as a fence between $a$ and $b$. Thus the fence insertion algorithm can infer more than one fence at a time, and the number of reachability queries can be decreased.

## 4 Experimental Results

Table 1 displays the results of running MEMORAX on several classical examples. For each of the examples, we give the total time for finding all minimal, sufficient sets of fences, using the methods SB and PB, with and without transforming the program

to register free form. In the table, "not-applicable" denotes that the corresponding approach is not applicable to the example. These correspond to applying a finite domain technique (SB and RFF) to an infinite domain program. Furthermore, out-of-mem is used to denote that the experiment failed to finish before consuming all available memory of the host computer. All examples were run on a laptop with a 2.27 GHz processor and 4 GB of memory.

| | Size Proc./States/ Var./Trans. | Total time seconds | | | | Fences necessary (smallest set) |
|---|---|---|---|---|---|---|
| | | SB | SB(rff) | PB | PB(rff) | |
| Simple Dekker | 2/6/2/6 | 0.0 | 0.0 | 0.0 | 0.0 | 1 per proc |
| Full Dekker | 2/22/3/28 | 0.4 | 0.2 | 0.1 | 0.1 | 1 per proc |
| Peterson | 2/12/3/14 | 1.9 | 1.0 | 3.5 | 0.4 | 1 per proc |
| Lamport Bakery (bounded) | 2/18/4/20 | out-of-mem | 61.2 | 152.7 | 17.9 | 2 per proc |
| Lamport Fast | 2/24/4/34 | 233.7 | 223.4 | 2.7 | 2.5 | 2 per proc |
| CLH Queue Lock | 2/30/4/42 | out-of-mem | 15.4 | out-of-mem | out-of-mem | 0 |
| Sense Reversing Barrier | 2/4/2/4 | 0.3 | 0.2 | 0.1 | 0.0 | 0 |
| Burns | 2/8/2/9 | 0.0 | 0.0 | 0.0 | 0.0 | 1 per proc |
| Dijkstra | 2/22/3/28 | out-of-mem | 0.4 | 1.0 | 2.0 | 1 per proc |
| Lamport Bakery (unbounded) | 2/18/4/20 | not-applicable | not-applicable | 166.2 | not-applicable | 2 per proc |
| Linux Ticket Lock (unbounded) | 2/4/2/4 | not-applicable | not-applicable | 0.4 | not-applicable | 0 |

**Table 1.** Experimental Results

## References


1. *MathSAT4*. http://mathsat4.disi.unitn.it/.
2. P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. `https://github.com/memorax/memorax`.
3. P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *SAS*, pages 164–180, 2012.
4. P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Counter-example guided fence insertion under tso. In *TACAS*, 2012.
5. S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12), 1996.
6. S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.
7. T. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *Formal Methods (FM)*, LNCS 4085. Springer, 2006.
8. M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FM-CAD*, 2011.
9. M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, 2011.
10. A. Linden and P. Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN*, 2010.
11. A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN*, 2011.
12. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP*. 2010.
13. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. In *Transactions on Programming Languages and Systems*, volume 10, pages 282–312. ACM, 1988.