Property-Driven Fence Insertion using Reorder Bounded Model Checking*

Saurabh Joshi and Daniel Kroening

Department of Computer Science
University of Oxford, UK
{saurabh.joshi,daniel.kroening}@cs.ox.ac.uk

Abstract. Modern architectures provide weaker memory consistency guarantees than sequential consistency. These weaker guarantees allow programs to exhibit behaviours where the program statements appear to have executed out of program order. Fortunately, modern architectures provide memory barriers (fences) to enforce the program order between a pair of statements if needed. Due to the intricate semantics of weak memory models, the placement of fences is challenging even for experienced programmers. Too few fences lead to bugs whereas overuse of fences results in performance degradation. This motivates automated placement of fences. Tools that restore sequential consistency in the program may insert more fences than necessary for the program to be correct. Therefore, we propose a property-driven technique that introduces *reorder-bounded exploration* to identify the smallest number of program locations for fence placement. We implemented our technique on top of CBMC; however, in principle, our technique is generic enough to be used with any model checker. Our experimental results show that our technique is faster and solves more instances of relevant benchmarks than earlier approaches.

1 Introduction

Modern multicore CPUs implement optimizations such as *store buffers* and *invalidate queues*. These features result in weaker memory consistency guarantees than sequential consistency (SC) [20]. Though such hardware optimizations offer better performance, the weaker consistency has the drawback of intricate and subtle semantics, thus making it harder for programmers to anticipate how their program might behave when run on such architectures. For example, it is possible for a pair of statements to appear to have executed out of the program order.

Consider the program given in Fig. 1a. Here, `x` and `y` are shared variables whereas `r1` and `r2` are thread-local variables. Statements s_1 and s_3 perform write operations. Owing to store buffering, these writes may not be reflected immediately in the memory. Next, both threads may proceed to perform the

* This research was supported by ERC project 280053 and by the Semiconductor Research Corporation (SRC) project 2269.002.

read operations s_2 and s_4. Since the write operations might still not have hit the memory, stale values for `x` and `y` may be read in `r2` and `r1`, respectively. This will cause the assertion to fail. Such behaviour is possible with architectures that implement *Total Store Order (TSO)*, which allows write-read reordering. Note that on a hypothetical architecture that guarantees sequential consistency, this would never happen. However, owing to store buffering, a global observer might witness that the statements are executed in the order (s_2, s_4, s_1, s_3), which results in the assertion failure. We say that (s_1, s_2) and (s_3, s_4) have been reordered.

Fig. 1b shows how the assertion might fail on architectures that implement *Partial Store Order (PSO)*, which permits write-write and write-read reordering. Using SC, one would expect to observe `r2` $==$ `1` if `r1` $==$ `1` has been observed. However, reordering of the write operations (s_1, s_2) leads to the assertion failure. Architectures such as Alpha, POWER and SPARC RMO even allow read-write and read-read reorderings, amongst other behaviours. Fortunately, all modern architectures provide various kinds of *memory barriers (fences)* to prohibit unwanted weakening. Due to the intricate semantics of weak memory models and fences, an automated approach to the placement of fences is desirable.

In this paper, we make the following contributions:

- We introduce *ReOrder Bounded Model Checking (ROBMC)*. In ROBMC, the model checker is restricted to exploring only those behaviours of a program that contain at most k reorderings for a given bound k. The reorder bound is a new parameter for bounding model checking that has not been explored earlier.
- We study how the performance of the analysis is affected as the bound changes.
- We implement two ROBMC-based algorithms. In addition, we implement earlier approaches in the same framework to enable comparison with ROBMC.

The rest of the paper is organized as follows. Section 2 provides an overview and a motivating example for ROBMC. Sections 3 and 4 provide preliminaries and describe earlier approaches respectively. ROBMC is described in Section 5. Related research is discussed in Section 6. Experimental results are given in Section 7. Finally, we make concluding remarks in Section 8.

2 Motivation and Overview

There has been a substantial amount of previous research on automated fence insertion [3, 4, 7, 11, 17, 23, 24]. We distinguish approaches that aim to restore sequential consistency (SC) and approaches that aim to ensure that a user-provided assertion holds. Since every fence incurs a performance penalty, it is desirable to keep the number of fences to a minimum. Therefore, a property-driven approach for fence insertion can result in better performance. The downside of the property-driven approach is that it requires an explicit specification.

Consider the example given in Fig. 1c. Here, `x,y,z,w` are shared variables initialized to 0. All other variables are thread-local. A processor that implements

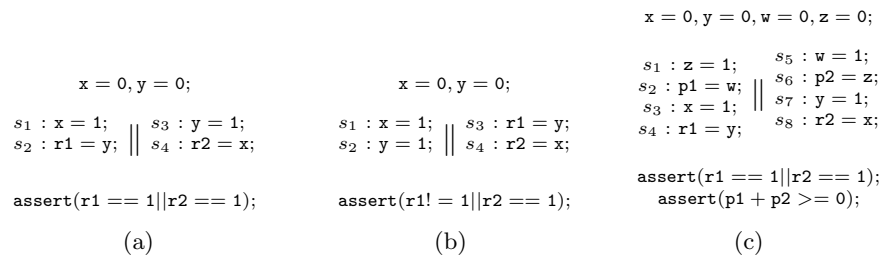


Fig. 1: (a) Reordering in TSO. (b) Reordering in PSO. (c) A program with *innocent* and *culprit* reorderings

total store ordering (TSO) permits a read of a global variable to precede a write to a different global variable when there are no dependencies between the two statements. Note that if (s_3, s_4) or (s_7, s_8) is reordered, the assertion will be violated. We shall call such pairs of statements *culprit pairs*. By contrast, the pairs (s_1, s_2) and (s_5, s_6) do not lead to an assertion violation irrespective of the order in which their statements execute. We shall call such pairs *innocent pairs*. A tool that restores SC would insert four fences, one for each pair mentioned earlier. However, only two fences (between s_3, s_4 and s_7, s_8) are necessary to avoid the assertion violation.

Some of the earlier property-driven techniques for fence insertion [3, 22] use the following approach. Consider a counterexample to the assertion. Every counterexample to the assertion must contain at least one culprit reordering. If we prevent all culprit reorderings, the program will satisfy the property. This is done in an iterative fashion. For all the counterexamples seen, a smallest set of reorderings S is selected such that S has at least one reordering in common with each of the counterexamples. Let us call such a set a *minimum-hitting-set* (*MHS*) over all the set of counterexamples C witnessed so far. All the weakenings in *MHS* are excluded from the program. Even though *MHS* may not cover all the culprit reorderings initially, it will eventually consist of culprit pairs only. Since one cannot distinguish the innocent pairs from the culprit ones a priori, such an approach may get distracted by innocent pairs, thus, taking too long to identify the culprit pairs.

To illustrate, let us revisit the example in Fig. 1c. Let us name the approach described above Fi (Fence Insertion). Let the first counterexample path π^1 be $(s_2, s_1, s_6, s_5, s_4, s_7, s_8, s_3)$. The set of reorderings is $\{(s_1, s_2), (s_3, s_4), (s_5, s_6)\}$. Method Fi may choose to forbid the reordering of $\{(s_1, s_2)\}$, as it is one of the choices for the *MHS*. Next, let $\pi^2 = (s_1, s_2, s_6, s_5, s_4, s_7, s_8, s_3)$. The set of reorderings for this trace is $\{(s_3, s_4), (s_5, s_6)\}$. There are multiple possible choices for *MHS*. For instance, Fi may choose to forbid $\{(s_5, s_6)\}$. Let $\pi^3 = (s_2, s_1, s_5, s_6, s_8, s_3, s_4, s_7)$. As the set of reorderings is $\{(s_1, s_2), (s_7, s_8)\}$, one of the choices for the *MHS* is $\{(s_1, s_2), (s_5, s_6)\}$. Recall that (s_1, s_2) and (s_5, s_6) are innocent pairs. On the other hand, (s_3, s_4) and (s_7, s_8) are culprit pairs.

Fi may continue with $\pi^4 = (s_1, s_2, s_5, s_6, s_4, s_7, s_8, s_3)$. The set of reorderings in π^4 is $\{(s_3, s_4)\}$. An adversarial *MHS* would be $\{(s_1, s_2), (s_3, s_4)\}$. Let π^5 be $(s_1, s_2, s_6, s_5, s_8, s_3, s_4, s_7)$. The reorderings $\{(s_5, s_6), (s_7, s_8)\}$ will finally lead to the solution $\{(s_3, s_4), (s_7, s_8)\}$. In the 6^{th} iteration Fi will find that the program is safe with a given *MHS*. For brevity, we have not considered traces with reorderings (s_1, s_4) and (s_5, s_8). In the worst case, considering these reorderings might lead to even more traces.

As we can see, the presence of innocent pairs plays a major role in how fast Fi will be able to find the culprit pairs. Consider a program with many more innocent pairs. Fi will require increasingly more queries to the underlying model checker as the number of innocent pairs increases.

To address the problem caused by innocent pairs, we propose *Reorder Bounded Model Checking* (ROBMC). In ROBMC, we restrict the model checker to exploring only the behaviours of the program that have at most k reorderings for a given reordering bound k. Let us revisit the example given in Fig. 1c to see how the bounded exploration affects the performance. Assume that we start with the bound $k = 1$. Since the model checker is forced to find a counterexample with only one reordering, there is no further scope for an innocent reordering to appear in the counterexample path. Let the first trace found be $\pi^1 = (s_1, s_2, s_4, s_5, s_6, s_7, s_8, s_3)$. There is only one reordering $\{(s_3, s_4)\}$ in this trace. The resulting *MHS* will be $\{(s_3, s_4)\}$. Let the second trace be $\pi^2 = (s_1, s_2, s_5, s_6, s_8, s_3, s_4, s_7)$. As the only reordering is $\{(s_7, s_8)\}$, the *MHS* over these two traces would be $\{(s_3, s_4)(s_7, s_8)\}$. The next query would declare the program safe. Now, even with a larger bound, no further counterexamples can be produced. This example shows how a solution can be found much faster with ROBMC compared to Fi. In the following sections, we describe our approach more formally.

3 Preliminaries

Let P be a concurrent program. A program execution is a sequence of events. An event e is a four-tuple

$$e \equiv \langle tid, in, var, type \rangle$$

where *tid* denotes the thread identifier associated with the event and *in* denotes the instruction that triggered the event. Instructions are dynamic instances of program statements. A program statement can give rise to multiple instructions due to loops and procedure calls. $stmt : Instr \rightarrow Stmt$ denotes a map from instructions to their corresponding program statements. The program order between any two instructions I_1 and I_2 is denoted as $I_1 <_{po} I_2$, which indicates that I_1 precedes I_2 in the program order. The component *var* denotes the global/shared variable that participated in the event e. The type of the event is represented by *type*, which can either be *read* or *write*. Without loss of generality, we assume that P only accesses one global/shared variable per statement.

Therefore, given a statement $s \in Stmt$, we can uniquely identify the global variable involved as well as the type of the event that s gives rise to. Any execution of program P is a sequence of events $\pi = (e_1, \ldots, e_n)$. The i^{th} event in the sequence π is denoted by $\pi(i)$.

Definition 1 *A pair of statements (s_1, s_2) of a program is said to be* reordered *in an execution π if:*

$$\begin{aligned}\exists_i \exists_j ((e_i.tid = e_j.tid) \wedge (\pi(i) = e_i) \wedge (\pi(j) = e_j) \\ \wedge (j < i) \wedge (e_i.in = I_1 \wedge e_j.in = I_2) \\ \wedge (I_1 <_{po} I_2) \wedge (stmt(I_1) = s_1 \wedge stmt(I_2) = s_2))\end{aligned}$$

According to Defn. 1, two statements are reordered if they give rise to events that occurred out of program order.

Definition 2 *We write $RO_A(s_1, s_2)$ to denote that an architecture A allows the pair of statements (s_1, s_2) to be reordered.*

Different weak memory architectures permit particular reorderings of events.

- **Total Store Order (TSO)**: TSO allows a read to be reordered before a write if they access different global variables.

$$RO_{tso}(s_1, s_2) \equiv (s_1.var \neq s_2.var) \wedge (s_1.type = write \wedge s_2.type = read)$$

- **Partial Store Order (PSO)**: PSO allows a read or write to be reordered before a write if they access different global variables.

$$RO_{pso}(s_1, s_2) \equiv (s_1.var \neq s_2.var) \wedge (s_1.type = write)$$

Partial-order based models for TSO, PSO, *read memory order (RMO)* and POWER are presented in detail in [7].

Definition 3 *Let C be a set consisting of non-empty sets $S_1, \ldots, S_n$. The set $\mathcal{H}$ is called a* hitting-set *(HS) of C if:*

$$\forall_{S_i \in C} \mathcal{H} \cap S_i \neq \emptyset$$

$\mathcal{H}$ is called a minimal-hitting-set *(mhs) if any proper subset of $\mathcal{H}$ is not a hitting-set. $\mathcal{H}$ is a* minimum-hitting-set *(MHS) of C if C does not have a smaller hitting-set. Note that a collection C may have multiple minimum-hitting-sets.*

4 Property-driven Fence Insertion

4.1 Overview

In this section we will discuss two approaches that were used earlier for property-driven fence insertion. We will present our improvements in the next section.

For a program P of size $|P|$, the total number of pairs of statements is $|P|^2$. Since the goal is to find a subset of these pairs, the search space is $2^{|P|^2}$. Thus, the search space grows *exponentially* as the size of the program is increased.

An automated method for fence insertion typically includes two components: (1) a model checker M and (2) a search technique that uses M iteratively in order to find a solution. We assume that the model checker M has the following properties:

- M should be able to find counterexamples to assertions in programs given a memory model.
- M should return the counterexample π in form of a sequence of events as described in Section 3.
- For a pair of statements (s_1, s_2) for which $RO_A(s_1, s_2)$ holds, M should be able to enforce an ordering constraint $s_1 \prec s_2$ that forbids the exploration of any execution where (s_1, s_2) is reordered.

4.2 Fence Insertion using Trace Enumeration

Alg. 1 is a very simple approach to placing fences in the program with the help of such a model checker. The algorithm is representative of the technique that is used in DFENCE [24]. Alg. 1 iteratively submits queries to M for a counterexample (Line 7). All the pairs of statements that have been reordered in π are collected in SP (Line 11). To avoid the same trace in future iterations, reordering of at least one of these pairs must be disallowed. The choice of which reorderings must be banned is left open. This process is repeated until no further error traces are found. Finally, *computeMinimalSolution*(ϕ) computes a minimal set of pairs of statements such that imposing ordering constraints on them satisfies ϕ.

Termination and soundness: Even though the program may have unbounded loops and thus potentially contains an unbounded number of counterexamples, Alg. 1 terminates. The reason is that an ordering constraint $s_1 \prec s_2$ disallows reordering of all events that are generated by (s_1, s_2). The number of iterations is bounded above by $2^{|P|^2}$, which is the size of the search space. Soundness is a consequence of the fact that the algorithm terminates only when no counterexamples are found. A minimal-hitting-set (mhs) is computed over all these counterexamples to compute the culprit pairs that must not be reordered. Since every trace must go through one of these pairs, it cannot manifest when the reordering of these pairs is banned. The number of pairs computed is minimal, thus, Alg. 1 does not guarantee the least number of fences. One can replace the minimal-hitting-set (mhs) with a minimum-hitting-set (MHS) in order to obtain such a guarantee.

Algorithm 1 Trace Enumerating Fence Insertion (Te)

1: **Input:** Program P
2: **Output:** Set S of pairs of statements that must not be reordered to avoid assertion failure
3: $C := \emptyset$
4: $S := \emptyset$
5: $\phi := true$
6: **loop**
7: $\quad \langle result, \pi \rangle := M(P_\phi)$
8: $\quad$ **if** $result = SAFE$ **then**
9: $\quad\quad$ **break**
10: $\quad$ **end if**
11: $\quad SP := GetReorderedPairs(\pi)$
12: $\quad$ **if** $SP = \emptyset$ **then**
13: $\quad\quad$ **print Error:** Program cannot be repaired
14: $\quad\quad$ **return errorcode**
15: $\quad$ **end if**
16: $\quad \phi := \phi \wedge \left(\bigvee_{(s_1, s_2) \in SP} s_1 \prec s_2 \right)$
17: **end loop**
18: $S := computeMinimalSolution(\phi)$
19: **return** S

Algorithm 2 Accelerated Fence Insertion (Fi)

1: **Input:** Program P
2: **Output:** Set S of pairs of statements that must not be reordered to avoid assertion failure
3: $C := \emptyset$
4: $S := \emptyset$
5: $\phi := true$
6: **loop**
7: $\quad \langle result, \pi \rangle := M(P_\phi)$
8: $\quad$ **if** $result = SAFE$ **then**
9: $\quad\quad$ **break**
10: $\quad$ **end if**
11: $\quad SP := GetReorderedPairs(\pi)$
12: $\quad$ **if** $SP = \emptyset$ **then**
13: $\quad\quad$ **print Error:** Program can not be repaired
14: $\quad\quad$ **return errorcode**
15: $\quad$ **end if**
16: $\quad C := C \cup \{SP\}$
17: $\quad S := MHS(C)$
18: $\quad \phi := \bigwedge_{(s_1, s_2) \in S} s_1 \prec s_2$
19: **end loop**
20: **return** S

4.3 Accelerated Fence Insertion

Alg. 2 is an alternative approach to fence insertion. The differences between Alg. 1 and Alg. 2 are highlighted. Alg. 2 has been used in [22,23] and is a variant of the approach used in [3]. Alg. 2 starts with an ordering constraint ϕ (Line 5), which is initially unrestricted. A call to the model checker M is made (Line 7) to check whether the program P under the constraint ϕ has a counterexample.

From a counterexample π, we collect the set of pairs of statements SP that have been reordered in π (Line 11). This set is put into a collection C. Next, we compute a minimum-hitting-set over C. This gives us the smallest set of pairs of statements that can avoid all the counterexamples seen so far. The original approach in [22] uses a minimal-hitting-set (mhs). The ordering constraint ϕ is updated using the minimum-hitting-set (Lines 17–18). Alg. 2 tells the model checker which reorderings from each counterexample are to be banned at every iteration, which is in contrast to Alg. 1. Alg. 2 assumes that an assertion violation in P is due to a reordering. If a counterexample is found without any reordering, the algorithm exits with an error (Lines 12–15). Finally, the algorithm terminates when no more counterexamples can be found (Lines 8–10).

Termination and soundness: The argument that applies to Alg. 1 can also be used to prove termination and soundness of Alg. 2. In addition, the constraint ϕ generated is generally stronger ($\phi_{Alg.\ 2} \rightarrow \phi_{Alg.\ 1}$) than the constraint generated by Alg. 1. Thus, for the same sequence of traces, Alg. 2 typically converges to a solution faster than Alg. 1.

5 Reorder-bounded Exploration

Alg. 2 can further be improved by avoiding innocent reorderings so that culprit reorderings responsible for the violation of the assertion are found faster.

As discussed in Section 2, Alg. 2 requires many iterations to converge and terminate in the presence of innocent reorderings. The reason is that the model checker may not return the simplest possible counterexample that explains the assertion violation due to reorderings. In order to address this problem, we need a model checker M' with an additional property as follows:

- M' takes P_ϕ and k as inputs. Here, P_ϕ is the program along with the ordering constraint ϕ and k is a positive integer. M' produces a counterexample π for P_ϕ such that π has at most k reorderings. If it cannot find a counterexample with at most k reorderings, then it will declare P_ϕ safe.

With a model checker M', we can employ Alg. 3 to speed up the discovery of the smallest set of culprit pairs of statements. The steps that differ from Alg. 2 in Alg. 3 are highlighted. Alg. 3 initializes the reordering bound k (Line 5) to a given lower bound K_1. The model checker M' is now called with this bound to obtain a counterexample that has at most k reorderings (Line 9). When the counterexample cannot be found, the bound k is increased according to some strategy denoted by *increaseStrategy* (Line 22). Note that collection C and the ordering constraint ϕ are preserved even when k is increased. Thus, when k is increased from k_1 to k_2, the search for culprit reorderings starts directly with the ordering constraints that repair the program for up to k_1 reorderings. Only those counterexamples that require more than k_1 and fewer than k_2 culprit reorderings will be reported. Let us assume that P does not have any counterexample with more than k_{opt} reorderings. If k_{opt} is much smaller than k, the performance of Alg. 3 might suffer due to interference from innocent reorderings. If the increase

Algorithm 3 ROBMC

1: **Input:** Program P, lower bound K_1 and an upper bound K_2
2: **Output:** Set S of pairs of statements that must not be reordered to avoid assertion failure
3: $C := \emptyset$
4: $S := \emptyset$
5: $k := K_1$
6: $\phi := true$
7: **while** $k \leq K_2$ **do**
8: **loop**
9: $\langle result, \pi \rangle := M'(P_\phi, k)$
10: **if** $result = SAFE$ **then**
11: **break**
12: **end if**
13: $SP := GetReorderedPairs(\pi)$
14: **if** $SP = \emptyset$ **then**
15: **print Error:** Program cannot be repaired
16: **return errorcode**
17: **end if**
18: $C := C \cup \{SP\}$
19: $S := MHS(C)$
20: $\phi := \bigwedge_{(s_1, s_2) \in S} s_1 \prec s_2$
21: **end loop**
22: $k := increaseStrategy(k)$
23: **end while**
24: **return** S

in k is too small, the algorithm might have to go through many queries to reach the given upper bound K_2. It can be beneficial to increase the bound k by a larger amount after witnessing a few successive *SAFE* queries, and by a smaller amount when a counterexample has been found recently.

Building M': A model checker M' that supports bounded exploration can be constructed from M as follows. For every pair (s_1, s_2) that can potentially be reordered, we introduce a new auxiliary Boolean variable a_{12}. Then, a constraint $\neg a_{12} \leftrightarrow (s_1 \prec s_2)$ can be added. This allows us to enforce the ordering constraint $s_1 \prec s_2$ by manipulating values assigned to a_{12}. For a given bound k, we can enforce a reorder-bounded exploration by adding a cardinality constraint $\sum a_{ij} \leq k$. This constraint forces only up to k auxiliary variables to be set to *true*, thus, allowing only up to k reorderings.

Optimizing Alg. 3: Even when the correct solution for the program is found, Alg. 3 has to reach the upper bound K_2 to terminate. This can cause many further queries for which the model checker M' is going to declare the program *SAFE*. To achieve soundness with Alg. 3, K_2 should be as high as the total number of all the pairs of statements that can be potentially reordered. This leads to a very high value for K_2, which may reduce the advantage that Alg. 3 has over Alg. 2. We can avoid these unnecessary queries if the model checker M' produces a proof whenever it declares the program P_ϕ as *SAFE*. This proof is analogous to an *unsatisfiable core* produced by many SAT/SMT solvers whenever

Algorithm 4 ROBMC-Et

```
1: Input: Program P, lower bound K_1 and an upper bound K_2
2: Output: Set S of pairs of statements that must not be reordered to avoid assertion failure
3: C := ∅
4: S := ∅
5: k := K_1
6: φ := true
7:  terminate := false
8: while  k ≤ K_2 and  terminate = false  do
9:    loop
10:      ⟨result, π, ψ⟩ := M'(P_φ, k)
11:      if  result = SAFE then
12:         if   not safeDueToBound(k, ψ)   then
13:            terminate := true
14:         end  if
15:         break
16:      end if
17:      SP := GetReorderedPairs(π)
18:      if  SP = ∅ then
19:         print  Error: Program cannot be repaired
20:         return  errorcode
21:      end if
22:      C := C ∪ {SP}
23:      S := MHS(C)
24:      φ := ⋀_{(s_1,s_2)∈S}  s_1 ≺ s_2
25:   end loop
26:   k := increaseStrategy(k)
27: end  while
28: return  S
```

the result of a query is *unsat*.[1] With this additional feature of M', we can check whether the cardinality constraint $\sum a_{ij} \leq k$ was the reason for declaring the program *SAFE*. If not, we know that P is safe under the ordering constraint ϕ irrespective of the bound. Therefore, Alg. 3 can terminate early as shown in Alg. 4. The difference between Alg. 3 and Alg. 4 is highlighted in Alg. 4. The model checker M' now returns ψ as a proof when P_ϕ is safe (Line 10). When M' declares P_ϕ as safe, Alg. 4 checks whether the bound k is the reason that P_ϕ is declared safe (Line 12). If not, the termination flag is set to *true* to trigger early termination (Line 13).

Termination and soundness: Let the program P have counterexamples with up to k_{opt} culprit reorderings. If the value of the upper bound K_2 for Alg. 3 and Alg. 4 is smaller than k_{opt}, there might exist traces that the algorithms fail to explore. For soundness, the value of K_2 should thus be higher than k_{opt}. Since k_{opt} is generally not known a priori, a conservative value of K_2 should be equal to the total number of pairs of statements for which reordering might happen ($RO_A(s_1, s_2)$ is *true*). Termination is guaranteed due to finiteness of the number of pairs of statements and K_2.

[1] SAT solvers such as MiniSat [13] and Lingeling [10] allow to query whether a given assumption was part of the unsatisfiable core [14].

6 Related work

There are two principal approaches for modelling weak memory semantics. One approach is to use operational models that explicitly model the buffers and queues to mimic the hardware [1, 2, 5, 11, 18, 23, 24]. The other approach is to axiomatize the observable behaviours using partial orders [6, 7, 9]. Buffer-based modelling is closer to the hardware implementation than the partial-order based approach. However, the partial-order based approach provides an abstraction of the underlying complexity of the hardware and has been proven effective [6]. Results on complexity and decidability for various weak memory models such as TSO, PSO and RMO are given in [8].

Due to the intricate and subtle semantics of weak memory consistency and the fences offered by modern architectures, there have been numerous efforts aimed at automating fence insertion [3, 4, 7, 11, 15, 17, 22–24]. These works can be divided into two categories. In one category, fences are inserted in order to restore sequential consistency [4,7,11]. The primary advantage is that no external specification is required. On the downside, the fences inferred by these methods may be unnecessary.

The second category are methods that insert only those fences that are required for a program to satisfy given properties [2, 3, 22–24]. These techniques usually require repetitive calls to a model checker or a solver. DFENCE is a dynamic analysis tool that falls into this category. Our work differs from DFENCE as ours is a fully static approach as compared to the dynamic approach used by DFENCE. A direct comparison with DFENCE cannot be made but we have implemented their approach in our framework and we present an experimental comparison using our re-implementation.

MEMORAX [3] and REMMEX [22, 23] also fall into the category of property-driven tools. MEMORAX [1] computes all possible minimal-hitting-set solutions. Though it computes the smallest possible solution, exhaustively searching for all possible solutions can make such an approach slow. Moreover, MEMORAX requires that the input program is written in RMM — a special purpose language. Alg. 2 captures what MEMORAX would do if it has to find only one solution. REMMEX also falls in the category of property-driven tools and their approach is given as Alg. 2.

Bounded model checking has been used for the verification of concurrent programs [6, 27]. In context-bounded model checking [19, 27], the number of interleavings in counterexamples is bounded, but executions are explored without depth limit. ROBMC is orthogonal to these ideas, as here the bound is on the number of event reorderings.

7 Implementation and Experimental Results

7.1 Experimental Setup

To enable comparison between the different approaches, we implemented all four algorithms in the same code base, using CBMC [6] as the model checker. CBMC

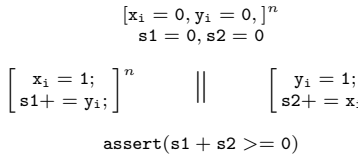


Fig. 2: A parameterized program. Here, $[\texttt{st}]^n$ denotes that the statement `st` is repeated n times.

explores loops until a given bound. Our implementation and the benchmarks used are available online at `http://www.cprover.org/glue` for independent verification of our results. The tool takes a C program as an input and assertions in the program as the specification.

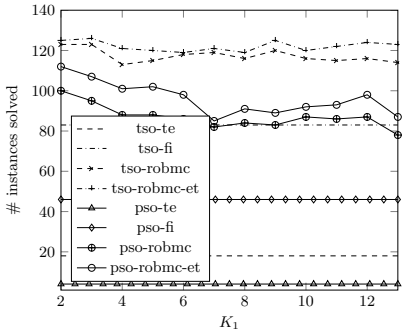
Alg. 1 closely approximates the approach used in DFENCE [24]. Alg. 2 resembles the approach used in REMMEX [22, 23] and a variant of MEMORAX [1, 3]. We used MINISAT 2.2.0 [13] as the SAT solver in CBMC. For all four algorithms incremental SAT solving is used. The cardinality constraints used in Alg. 3 and Alg. 4 are encoded incrementally [25]. Thus, the program is encoded only once while the ordering constraints are changed in every iteration using the assumption interface of the solver. The experiments were performed on a machine with 8-core Intel Xeon processors and 48 GB RAM. The *increaseStrategy*(k) used for algorithms Alg. 3 and Alg. 4 doubles the bound k.

7.2 Benchmarks

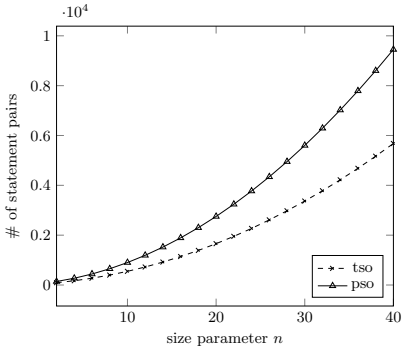
Mutual exclusion algorithms such as *dekker*, *peterson* [26], *lamport* [21], *dijkstra* [21] and *szymanski* [28] as well as *ChaseLev* [12] and *Cilk* [16] work stealing queues were used as benchmarks. All benchmarks have been implemented in C using the `pthread` library. For mutual exclusion benchmarks, a shared counter was added and incremented in the critical section. An assertion was added to check that none of the increments are lost. In addition, all the benchmarks were augmented with a parametric code fragment shown in Fig. 2, which increases the number of innocent pairs as n is increased. The parameter n was increased from 2 to 40 with an increment of 2. Thus, each benchmark has 20 parametric instances, which makes the total number of problem instances for one memory model 140.

7.3 Results

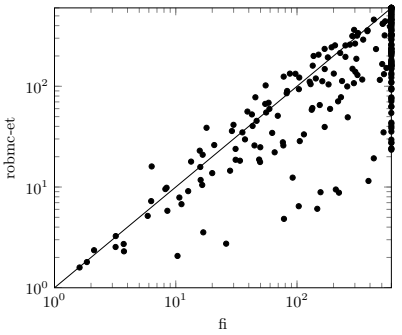
We ran our experiments for the TSO and PSO memory models for all the instances with the timeout of 600 seconds. From now on, we will refer to Alg. 1 as TE, Alg. 2 as FI, Alg. 3 as ROBMC and Alg. 4 as ROBMC-ET. In our experiments we found that all algorithms produce the smallest set of fence placement for every problem instance. Thus, we will focus our discussion on the relative performance of these approaches.



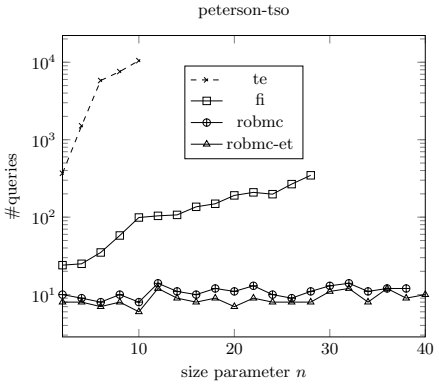
(a) # of instances solved



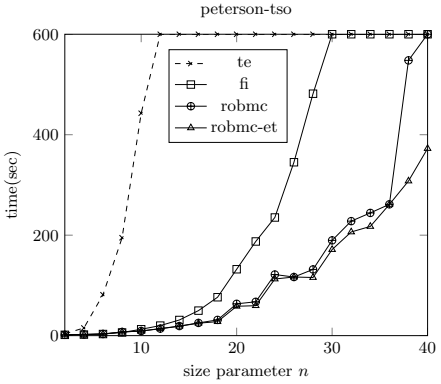
(b) # of statement pairs



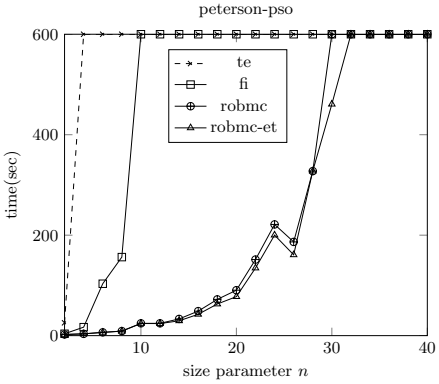
(c) ROBMC-Et (with $K_1 = 5$) v/s Fi



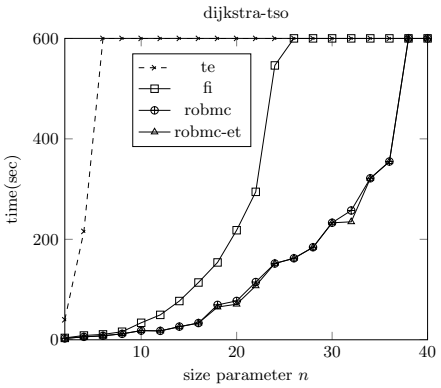
(d) peterson on TSO ($K_1 = 5$)



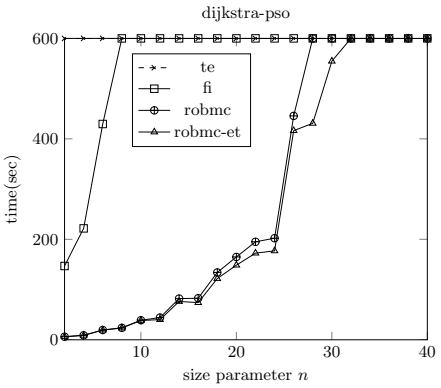
(e) peterson on TSO ($K_1 = 5$)



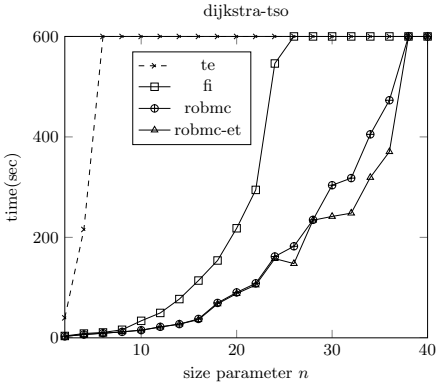
(f) peterson on PSO ($K_1 = 5$)



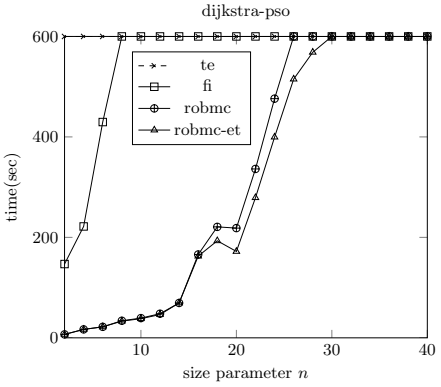
(g) dijkstra on TSO ($K_1 = 5$)



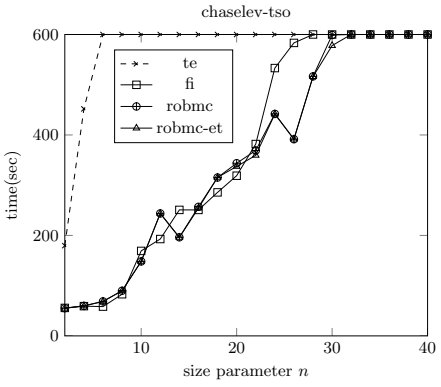
(h) dijkstra on PSO ($K_1 = 5$)



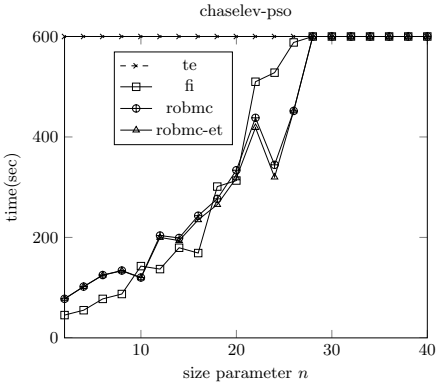
(i) dijkstra on TSO ($K_1 = 10$)



(j) dijkstra on PSO ($K_1 = 10$)



(k) ChaseLev on TSO ($K_1 = 5$)



(l) ChaseLev on PSO ($K_1 = 5$)

Fig. 3: For all experiments : Timeout=600 seconds, K_2=all pairs of statement (for soundness)

Fig. 3a shows the effect of changing the value of the parameter K_1 in ROBMC and ROBMC-Et. Remember that the bound is increased gradually from K_1 to K_2. Here, K_2 is always set to the total number of statement pairs in the program to guarantee soundness. Te and Fi do not have a parameter K_1, and thus, their corresponding plots are flat. Fig. 3a shows that ROBMC and ROBMC-Et solve far more instances than Te and Fi. The gap is even wider for the PSO memory model, which allows more reordering, and thus the number of innocent pairs are significantly higher compared to TSO on the same program. As expected, ROBMC-Et performs better, due to the early termination optimization. The value of K_1 barely affects the number of solved instances. The moderate downward trend for the plots as K_1 increases suggests that as K_1 increases, ROBMC tends to behave more and more like Fi.

Fig. 3b shows the increase in the total number of statement pairs that can potentially be reordered as the parameter n (Fig. 2) increases for the Peterson algorithm. As expected, the number of pairs grows quadratically in n. For PSO, the increase is steeper, as PSO allows more reordering than TSO. This explains the better performance of the ROBMC approaches on PSO.

The log-scale scatter plot in Fig. 3c compares the run-time of ROBMC-Et with $K_1 = 5$ with Fi over all 280 problem instances. Fi times out significantly more often (data points where both time out are omitted). Even on the instances solved by both the approaches, ROBMC-Et clearly outperforms Fi on all but a few instances. Those instances where Fi performs better typically have very few innocent pairs. Note that the queries generated by ROBMC-Et are more expensive, as our current implementation uses cardinality constraints to enforce boundedness. Thus, it is possible for Fi to sometimes perform better even though it generates a larger number of queries to the underlying model checker.

The semi-log-scale plot in Fig. 3d gives the number of queries to the model checker required by the approaches for the peterson algorithm on TSO. Te and Fi generate exponentially many queries to the model checker as n increases. By contrast, the number of queries generated by ROBMC and ROBMC-Et virtually remains unaffected by n. This is expected as the search is narrow and focussed owing to the bound k.

Fig. 3e and Fig. 3f give the relative performance of all the algorithms when the size and number of innocent pairs increases with the parameter n. All plots show an exponential trajectory, indicating that ROBMC does not fundamentally reduce the complexity of the underlying problem. Even though the number of queries required remains constant (Fig. 3d), each such query becomes more expensive because of the cardinality constraints.

However, the growth rate for ROBMC and ROBMC-Et is much slower compared to Te and Fi. Fig. 3e and Fig. 3f corroborate the claim that ROBMC-based approaches perform much better when there are a significant number of innocent pairs. For PSO, the performance gained by using ROBMC is even higher, as PSO allows more reordering. Similar trends are observed for dijkstra algorithm in Figs. 3g and 3h. Plots in Figs. 3g and 3i as well as Figs. 3h and 3j show that the performance of ROBMC-based approaches is not highly sensitive to the

value of K_1 as it changes from 5 to 10. This is consistent with the observation made from Fig. 3a.

The performance comparision for the ChaseLev work stealing queue is given in Figs. 3k and 3l. Here it can be seen that the threshold (in terms of innocent pairs) needed for ROBMC to surpass other approaches is higher. Even for such a case, ROBMC still provides competitive performance when the number of innocent pairs are low. ROBMC regains its superiority towards the end as the number of innocent pairs increases. Thus, even when every individual query is more expensive (due to the current implementation that uses cardinality constraints to enforce the bound), ROBMC always provides almost equal or better performance for all the benchmarks.

8 Concluding Remarks

ROBMC is a new variant of Bounded Model Checking that has not been explored before. Our experimental results indicate substantial speedups when applying ROBMC for the automated placement of fences on programs with few culprit pairs and a large number of innocent pairs. In particular, we observe that the speedup obtained by using ROBMC increases when targeting a weaker architecture. Thus, ROBMC adds a new direction in bounded model checking which is worth exploring further.

Acknowledgement The authors would like to thank Vincent Nimal for helpful discussions on the related work.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7214, pp. 204–219. Springer (2012)
2. Abdulla, P.A., Atig, M.F., Lang, M., Ngo, T.P.: Precise and sound automatic fence insertion procedure under PSO. In: Networked Systems (NETYS). Springer (2015)
3. Abdulla, P., Atig, M., Chen, Y.F., Leonardsson, C., Rezine, A.: MEMORAX, a precise and sound tool for automatic fence insertion under TSO. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7795, pp. 530–536. Springer (2013)
4. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence: A static analysis approach to automatic fence insertion. In: Computer Aided Verification (CAV). LNCS, vol. 8559, pp. 508–524. Springer (2014)
5. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: European Symposium on Programming (ESOP). LNCS, vol. 7792, pp. 512–532. Springer (2013), `http://dx.doi.org/10.1007/978-3-642-37036-6_28`
6. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification (CAV), LNCS, vol. 8044, pp. 141–157. Springer (2013)

7. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models (extended version). Formal Methods in System Design (FMSD) 40(2), 170–205 (2012)
8. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL. pp. 7–18. ACM (2010)
9. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Principles of Programming Languages (POPL). pp. 55–66. ACM (2011)
10. Biere, A.: Lingeling. http://fmv.jku.at/lingeling/
11. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: European Symposium on Programming (ESOP). LNCS, vol. 7792, pp. 533–553. Springer (2013)
12. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 21–28. ACM (2005)
13. Eén, N., Sörensson, N.: MiniSat. http://minisat.se/Main.html
14. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003)
15. Fang, X., Lee, J., Midkiff, S.P.: Automatic fence insertion for shared memory multiprocessing. In: ICS. pp. 285–294 (2003)
16. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Programming Language Design and Implementation (PLDI). pp. 212–223 (1998)
17. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD. pp. 111–120. Austin, TX (2010)
18. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: Programming Language Design and Implementation (PLDI). pp. 187–198. ACM (2011)
19. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Computer Aided Verification (CAV). LNCS, vol. 5123, pp. 37–51. Springer (2008)
20. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)
21. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst. 5(1), 1–11 (1987)
22. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in relaxed memory systems. In: Model Checking Software (SPIN), LNCS, vol. 6823, pp. 144–160. Springer (2011)
23. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7795, pp. 339–353. Springer (2013)
24. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., Yahav, E.: Dynamic synthesis for relaxed memory models. In: Programming Language Design and Implementation (PLDI). pp. 429–440. ACM (2012)
25. Martins, R., Joshi, S., Manquinho, V.M., Lynce, I.: Incremental cardinality constraints for MaxSAT. In: CP. LNCS, vol. 8656, pp. 531–548. Springer (2014)
26. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. 12(3), 115–116 (1981)
27. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 3440, pp. 93–107. Springer (2005)

28. Szymanski, B.K.: A simple solution to Lamport's concurrent programming problem with linear wait. In: ICS. pp. 621–626. ACM (1988)