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 overseer 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. For a parametric program that increases in size, our technique not only solves more instances but also yields a 17x speedup for the largest instance solved by an earlier approach. We report experimental results on relevant benchmarks and compare with earlier approaches.

I. 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) [24]. 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 on such architectures.

Consider the program given in Fig. 1a. Here, x and y are shared variables whereas r1 and r2 are thread-local variables. Statements s1 and s3 perform write operations, however, owing to store buffering, these writes may not be reflected immediately in the memory. Next, both threads may proceed to perform the read operations s2 and s4. 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, due to store buffering, a global observer might witness that the statements are executed in the order (s2, s4, s1, s3), which results in the assertion failure. We say that (s1, s2) and (s3, s4) 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 (s1, s2) would lead 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 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). It is a new parameter for bounding model checking that has not been explored earlier. In ROBMC, the model checker is restricted to explore those behaviours of a program that contain at most k reorderings for a given bound k.
- 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 II provides an overview and a motivating example for ROBMC. Sections III and IV provide preliminaries and describe earlier approaches respectively. ROBMC is described in Section V. Related research is discussed in Section VI. Experimental results are given in Section VII. Finally, we make concluding remarks in Section VIII.

II. MOTIVATION AND OVERVIEW

There has been a substantial amount of previous research on automated fence insertion [2], [3], [5], [10], [21], [27], [28]. 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 total store ordering (TSO) permits a read of a 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. On the other hand, 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 [2], [26] use the following approach. Consider a counterexample to the assertion. Every counterexample to the assertion must contain at least one culprit reordering. If one prevents all culprit reorderings, the program will satisfy the property. This is done in an iterative fashion. For all the counterexamples seen, a minimal 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 minimal-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 \( \pi^1 \) be \((s_2, s_1, s_0, s_5, s_4, s_7, s_8, s_3)\). 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 \( \pi^4 \) is \({(s_3, s_4)}\). An adversarial \( mhs \) would be \({(s_1, s_2), (s_3, s_4)}\). Let \( \pi^2 \) 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 6th 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. Also, at each step, we have used the minimum-hitting-set (MHS) instead of a minimal-hitting-set. If an adversarial \( mhs \) is used at each iteration, many more iterations might be required.

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 explore only those 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.
III. Preliminaries

Let $P$ be a concurrent program. A program execution is a sequence of events. An event $e$ is a four-tuple $e ≡ (tid, in, var, type)$ 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. $stmt : Instr → Stmt$ denotes a map from instructions to their corresponding program statements. The program order between any two instruction $I_1$ 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 $s ∈ Stmt$, 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 $π = (e_1, . . . , e_n)$. The $i$th event in the sequence $π$ is denoted by $π(i)$.

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

$$(e_i.tid = e_j.tid) ∧ (π(i) = e_i) ∧ (π(j) = e_j)$$
$$∧ (j < i) ∧ (e_i.in = I_1 ∧ e_j.in = I_2)$$
$$∧ (I_1 ≺_{po} I_2) ∧ (stmt(I_1) = s_1 ∧ stmt(I_2) = s_2)$$

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

Definition 2: Two statements $s_1$ and $s_2$ are said to be related via a dependence relation denoted as $s_1 ≺_{dp} s_2$ if $s_2$ is control or data dependent on $s_1$.

Definition 3: $RO_A(s_1, s_2)$ denotes that an architecture $A$ allows $(s_1, s_2)$ 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 and are independent of each other.

\[ RO_{ts}(s_1, s_2) ≡ (s_1.var ≠ s_2.var) ∧ (s_1 ≠_{dp} s_2) \]
$$∧ (s_1.type = write ∧ s_2.type = read)$$

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

\[ RO_{ps}(s_1, s_2) ≡ (s_1.var ≠ s_2.var) ∧ (s_1 ≠_{dp} s_2) \]
$$∧ (s_1.type = write)$$

Partial-order based models for TSO, PSO, read memory order (RMO) and POWER are presented in detail in [5].

Definition 4: Let $C$ be a set consisting of non-empty sets $S_1, . . . , S_n$. The set $H$ is called a hitting-set (HS) of $C$ if:

$$∀ s_i ∈ C \cap S_i ≠ \emptyset$$

$H$ is called a minimal-hitting-set (mhs) if any proper subset of $H$ is not a hitting-set. $H$ 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.

IV. Property-driven fence insertion

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

An automated fence insertion approach typically includes two components: (1) a model checker $M$ and (2) a search technique that uses $M$ iteratively in order to find a solution. 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. 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 III.
- 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 $s_1 ≺ s_2$ that forbids the exploration of any execution where $(s_1, s_2)$ is reordered.

Alg. 1 is a very simple approach to placing fences in the program with the help of such a model checker. Alg. 1 is representative of the technique that has been used in DFENCE [28]. 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 of these reorderings must be banned is left open. This process is repeated until no further error traces are found.

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 $s_1 ≺ s_2$ disallows reordering of all events that are generated by $(s_1, s_2)$. Since the search space is $2^{|P|^2}$, the number of iterations is bounded above by it. Soundness is a consequence of the fact that the algorithm terminates only when no counterexamples are found. A minimal-hitting-set 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 with a minimum-hitting-set 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: $\phi = \text{true}$
6: loop
7: $(\text{result, } \pi) = M(P_\phi)$
8: if result $== \text{SAFE}$ then
9: break
10: end if
11: $SP = \text{GetReorderedPairs}(\pi)$
12: if $SP == \emptyset$ then
13: print Error: Program cannot be repaired
14: return ErrorCode
15: end if
16: $\phi = \phi \land \bigvee_{(s_1, s_2) \in SP} s_1 \prec s_2$
17: end loop
18: $S = \text{computeMinimalSolution}(\phi)$
19: return $S$

Algorithm 2: 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: $\phi = \text{true}$
6: loop
7: $(\text{result, } \pi) = M(P_\phi)$
8: if result $== \text{SAFE}$ then
9: break
10: end if
11: $SP = \text{GetReorderedPairs}(\pi)$
12: if $SP == \emptyset$ then
13: print Error: Program cannot be repaired
14: return ErrorCode
15: end if
16: $C = C \cup \{SP\}$
17: $S = \text{MHS}(S)$
18: $\phi = \bigwedge_{(s_1, s_2) \in S} s_1 \prec s_2$
19: end loop
20: return $S$

Alg. 2 is a different approach to fence insertion. The difference to Alg. 1 is highlighted. Alg. 2 has been used in [26], [27] and is a variant of the approach used in [2]. Alg. 2 starts with an ordering constraint $\phi$ (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 $\phi$ has a counterexample. From a counterexample $\pi$, we collect the set of pairs of statements $SP$ that have been reordered in $\pi$ (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 [26] uses a minimal-hitting-set (mhs). The ordering constraint $\phi$ 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 be used to prove termination and soundness of Alg. 2. In addition, the constraint $\phi$ generated is generally stronger ($\phi_{\text{Alg. 2}} \rightarrow \phi_{\text{Alg. 1}}$) 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.

V. REORDER-BOUNDED EXPLORATION

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

As discussed in Section II, 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 $P_\phi$ and $k$ as inputs. Here, $P_\phi$ is the program along with the ordering constraint $\phi$ and $k$ is a positive integer. $M'$ produces a counterexample $\pi$ for $P_\phi$ such that $\pi$ has at most $k$ reorderings. If it cannot find a counterexample with at most $k$ reorderings, then it will declare $P_\phi$ 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 $\phi$ 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 $k_{\text{opt}}$ reorderings. If $k_{\text{opt}}$ is much smaller than $k$, the performance of Alg. 3 might suffer due to interference from innocent reorderings. If the increase in $k$ is too small, the algorithm might have to go through a many queries to reach the given upper bound $K_2$. It can be beneficial to increase the bound $b$ 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 $\neg a_{12} \rightarrow (s_1 \prec s_2)$ can be added. This allows us to enforce the ordering constraint $s_1 \prec s_2$ by manipulating values assigned to
Algorithm 3: ROBMC

1: \textbf{Input: } Program \( P \), lower bound \( K_1 \) and an upper bound \( K_2 \)
2: \textbf{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: \( \phi = \text{true} \)
7: \textbf{while } \( k \leq K_2 \) and \( \text{terminate} == \text{false} \) \textbf{do}
8: \begin{align*}
9: & (\text{result, } \pi) = M'(P_\phi, k) \\
10: & \text{if } \text{result} == \text{SAFE} \text{ then break} \\
11: & \text{end if}
12: & \text{end if}
13: & \text{if } SP = \text{GetReorderedPairs}(\pi) \text{ then} \\
14: & \quad \text{print Error: Program cannot be repaired} \\
15: & \quad \text{return } \text{errorcode}
16: & \text{end if}
17: & C = C \cup \{ SP \}
18: & S = \text{MHS}(C)
19: & \phi = \exists (s_1, s_2) \in S \ s_1 \prec s_2
20: \end{align*}
21: \textbf{end while}
22: \( k = \text{increaseStrategy}(k) \)
23: \textbf{return } S

Algorithm 4: ROBMC-ET

1: \textbf{Input: } Program \( P \), lower bound \( K_1 \) and an upper bound \( K_2 \)
2: \textbf{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: \( \phi = \text{true} \)
7: \( \text{terminate} = \text{false} \)
8: \textbf{while } \( k \leq K_2 \) and \( \text{terminate} == \text{false} \) \textbf{do}
9: \begin{align*}
10: & (\text{result, } \pi, \psi) = M'(P_\phi, k) \\
11: & \text{if } \text{result} == \text{SAFE} \text{ then} \\
12: & \quad \text{if } \not\text{safeDueToBound}(k, \psi) \text{ then} \\
13: & \quad \quad \text{terminate} = \text{true} \\
14: & \quad \text{end if}
15: & \text{end if}
16: & \text{break}
17: & \text{end if}
18: & \text{if } SP = \text{GetReorderedPairs}(\pi) \text{ then} \\
19: & \quad \text{print Error: Program cannot be repaired} \\
20: & \quad \text{return } \text{errorcode}
21: & \text{end if}
22: & C = C \cup \{ SP \}
23: & S = \text{MHS}(C)
24: & \phi = \exists (s_1, s_2) \in S \ s_1 \prec s_2
25: \end{align*}
26: \textbf{end while}
27: \( k = \text{increaseStrategy}(k) \)
28: \textbf{return } S

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 \text{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 \text{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 \( P_\phi \) as \text{SAFE}. This proof is analogous to an unsatisfiable core produced by many SAT/SMT solvers whenever the result of a query is \text{unsat}. With this additional feature of \( M' \), we can check whether the cardinality constraint \( \sum a_{ij} \leq k \) was the reason of the program being declared \text{SAFE}. If not, we know that \( P \) is safe under the ordering constraint \( \phi \) irrespective of the bound. Therefore, Alg. 3 can terminated 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 \( \psi \) as a proof when \( P_\phi \) is safe (Line 10). When \( M' \) declares \( P_\phi \) as safe, Alg. 4 checks whether the bound \( k \) is the reason that \( P_\phi \) is declared safe (Line 12). If not, the termination flag is set to \text{true} to trigger early termination (Line 13).

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

**VI. 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], [10], [22], [27], [28]. The other approach is to axiomatize the observable behaviours using partial orders [4], [5], [7]. Buffer-based modelling is closer to the hardware implementation than the partial-order based approach. However, the partial-order based approach provides an abstraction over the underlying complexity of the hardware and has been proven effective [4].

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 [2], [3], [5], [10], [18], [21], [26]–[28]. These works can be divided into two categories. In one category, fences are inserted in order to restore sequential consistency [3], [5], [10]. The primary advantage is that no external specification is required. On the downside, the fences inferred by these methods may be unnecessary for the program to be correct.

The second category are methods that insert only those fences that are required for a program to satisfy given properties [2], [26]–[28]. These techniques usually require repetitive calls to a model checker or a solver. DFENCE is a dynamic SAT solvers such as MiniSat [16] and Lingeling [8] allow to query whether a given assumption was part of the unsatisfiable core [17].
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, however, we have implemented their approach in our framework and we present an experimental comparison using our re-implementation.

MEMORAX [2] and REMMEX [26], [27] also fall into the category of property-driven tools. MEMORAX [1] computes all possible minimal-hitting-set solutions. Though it contains 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.

Results on complexity and decidability for various weak memory models such as TSO, PSO and RMO are given in [6]. Subtle changes done by innocuous-looking program transformations and their effects under weak memory models have been studied in [12].

Bounded model checking has been used for the verification of hardware [9], sequential programs [14] and concurrent programs [4], [30]. In context-bounded model checking [23], [30], 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.

VII. IMPLEMENTATION AND EXPERIMENTS

Experimental Setup: To enable comparison between the different approaches, we implemented all four algorithms in the same code base, using Cbmc [4] as the model checker. Our implementation and the benchmarks used are available online at [20] for independent verification of our results. Alg. 1 closely approximates the approach used in DFENCE [28]. Alg. 2 resembles the approach used in REMMEX [26], [27] and a variant of MEMORAX [1], [2]. We use LINGELING [8] as the SAT solver in CBMC. For all four algorithms incremental SAT solving is used. 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. For Alg. 3 and Alg. 4, the value of $K_2$ was set to the total number of statement pairs in the program to guarantee soundness. The increaseStrategy($k$) used for both these algorithms doubles the bound $k$. Our implementation takes a C program as an input and assertions in the program capture the specification. All the benchmarks have been implemented in C using the pthreadd library. For all benchmarks that implement a form of mutual exclusion, a shared counter was added and incremented in the critical section. An assertion was added to check that none of the increments are lost.

Parametric Benchmarks: Fig. 2a gives a parametric program that increases in size as the input parameter $n$ grows. Note that the statements inside the square brackets form innocent pairs. Only the two pairs outside the brackets are culprit pairs. A tool that restores SC would insert $2n$ additional fences that are not needed for the program to satisfy the assertion. Fig. 3a gives the time taken by the algorithms as the parameter $n$ is increased in Fig. 2a. As expected, the advantage of ROBMC magnifies as the number of innocent pairs increases. The exponential nature of the search space can be seen in the plots. Fig. 3b gives the number of queries that the algorithms submit to the model checker. Alg. 1 performs the worst, as the number of queries quickly increases and reaches a few thousand. The number of queries generated by Alg. 3 and Alg. 4 hardly varies with $n$. The reason is that these algorithms discover the culprit pairs quickly and do not get distracted by innocent pairs. However, Fig. 3a shows that each such query increasingly takes more time. We implemented ROBMC using cardinality constraints as described in Section V. This in turn pushes the search for culprit pairs into the SAT solver. Thus, as the size increases, the SAT solver will require more time per query.

Fig. 2b gives a parametric program where the number of
Fig. 3: Effect of ROBMC on parametric programs of Fig. 2. $K_1 = 2$ and $K_2$ is always kept equal to the total number of pairs to retain soundness for Alg. 3 and Alg. 4. Time out = 600 seconds.

threads is increased. Here, all pairs are culprits. As can be seen in Fig. 3c, the benefit of Alg. 3 and Alg. 4 is minor when there are no innocent pairs. The plots for Alg. 3 and Alg. 4 are difficult to distinguish as their performance is very close. Fig. 3d illustrates how the number of queries to the model checker increases as the number of threads grows. All the algorithms witness an increase because the number of culprit pairs increases with the number of threads.

Fig. 4 shows that Alg. 3 and Alg. 4 are sensitive to the value of $K_1$. In our implementation, ROBMC is realized using cardinality constraints. Let $R$ be the total number of pairs to be considered. If $K_1$ is too small, the solver does not have many options for reordering. Thus, to find a satisfying assignment, it has to try most of the $\binom{R}{K_1}$ possibilities within the given bound. Therefore, each query is computationally expensive. For a larger $K_1$, it is easier to find a satisfying assignment, but on the other hand, size of the formula that encodes the cardinality constraint increases to $O(RK_1)$ [31]. This implies that there is a “sweet spot” for the choice of $K_1$. For every program, the ideal range for $K_1$ varies depending on the size and the complexity of the program. We have observed that this range is large for most programs. Multiple instances can be run in parallel in portfolio-style with different values of $K_1$, and the algorithm stops once any instance returns a result.

**Benchmarks from Related Work:** Tab. 1 compares the performance of the algorithms on the benchmarks typically used in related work. These benchmarks are generally small, and contain only few innocent pairs, if any. All algorithms are sound and compute the smallest set of statement pairs that should be prevented from being reordered to fix the program. With the single exception of szymanski on PSO, Alg. 3 and Alg. 4 outperform Alg. 1 and Alg. 2 substantially. It is worth noting that the order in which the counterexamples are produced plays a crucial role in the performance of all four algorithms.

**VIII. Concluding Remarks**

ROBMC is a new variant of Bounded Model Checking that has not been explored before. Our experimental results highlight the advantages of ROBMC for automated placement of fences on programs with few culprit pairs and a large number of innocent pairs. Our implementation of ROBMC is sensitive to the reordering bound $K_1$ due to the way we have implemented it at present. In the future, it would be interesting...
TABLE I: \#Pairs=Total number of pairs of statements, \#Q = Queries to the model checker, T (sec)=Time in seconds, \(K_1\)=value of \(K_1\) for which the algorithm took the least amount of time, TimeOut=600 seconds

(a)

| Benchmark  | \#Pairs | #Q (T (sec)) | TSO | #Q (T (sec)) | \(K_1\) | ROBMC | \(K_1\) | ROBMC-ET |
|------------|---------|--------------|-----|--------------|--------|-------|--------|----------|
| dekker     | 78      | 403          | 2.867 | 9.335       | 11     | 5.833 | 19     | 5.696    |
| peterson   | 40      | 85           | 23   | 3.763        | 7      | 3.201 | 19     | 2.867    |
| szymanski  | 94      | 2464         | 2464 | 111.135      | 15     | 39.234| 30     | 30.974   |
| dijkstra   | 82      | 1918         | 403  | 47.966       | 18     | 13.913| 18     | 13.505   |
| lamport    | 132     | 2856         | 2.721 | 851          | 102    | 103.59| 5      | 81.593   |
| ChaseLevWSQ | 57    | 101          | 130.827 | 162.174 | 4     | 160.812| 27     | 106.58   |
| CilkWSQ    | 124     | 26           | 57    | 21.228       | 5      | 50.862| 12     | 48.789   |

(b)

| Benchmark  | \#Pairs | #Q (T (sec)) | PSO | #Q (T (sec)) | \(K_1\) | ROBMC | \(K_1\) | ROBMC-ET |
|------------|---------|--------------|-----|--------------|--------|-------|--------|----------|
| dekker     | 100     | 1793         | 176.836 | 18.386      | 145    | 8.48  | 4      | 16.502   |
| peterson   | 53      | 450          | 53.386 | 5.271        | 58     | 2.721 | 30     | 2.335    |
| szymanski  | 126     | 21027        | 281.279 | 29.854      | 56     | 29.854| 23     | 35.463   |
| dijkstra   | 145     | –            | 176.836 | –           | –      | –     | –      | –        |
| lamport    | 193     | –            | 851   | 203.302      | 8      | 47    | 15     | 77.101   |
| ChaseLevWSQ | 96    | –            | 130.827 | 156.85      | 30     | 5     | 29     | 130.378  |
| CilkWSQ    | 124     | 264          | 63.062 | 49.373       | 10     | 49.373| 17     | 47.725   |

to explore alternative means to achieve ROBMC that are less sensitive to the choice \(K_1\).

ACKNOWLEDGEMENT

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

REFERENCES

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 7214 of LNCS, pages 204–219. Springer, 2012.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In TACAS, volume 7795 of LNCS, pages 530–536, 2013.

[3] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don’t sit on the fence: A static analysis approach to automatic fence insertion. In CAV, LNCS. Springer, 2014. Available on arXiv.

[4] Jade Alglave, Daniel Kroening, and Michael Tautsching. Partial orders for efficient bounded model checking of concurrent software. In CAV, volume 8044 of LNCS, pages 141–157, 2013.

[5] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). FMSD, 40(2):170–205, 2012.

[6] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In CAV, LNCS. Springer, 2014. Available on arXiv.

[7] Leslie Lamport. Solution of a problem in concurrent programming control. Commun. ACM, 8(9):569–, September 1965.

[8] Niklas Eén and Niklas Sörensson. Minisat. http://minisat.se/Main.html.

[9] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci., 89(4):543–560, 2003.

[10] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In ICS, pages 285–294, 2003.

[11] Matteo Frigo, Charles E. Leiserson, and Keith B. Randall. The implementation of the Cilk-5 multithreaded language. In PLDI, pages 212–223, 1998.

[12] Saurabh Joshi and Daniel Kroening. CBMC-Repair. http://www.cprover.org/gheu.

[13] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In FMCAD, pages 111–120, Austin, TX, 2010.

[14] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In PLDI, pages 187–198, 2011.

[15] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under TSO. In TACAS, pages 193–207, 2013.

[16] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers, 28(9):690–691, 1979.

[17] Leslie Lamport. A fast mutual exclusion algorithm. ACM Trans. Comput. Syst., 5(1):1–11, 1987.

[18] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In Model Checking Software, volume 6823 of LNCS, pages 144–160, Springer, 2011.

[19] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in PSO memory systems. In TACAS, volume 7795 of LNCS, pages 339–353, 2013.
[28] Feng Liu, Nayden NedeV, Nedyalko Prisadnikov, Martin Vechev, and
Eran Yahav. Dynamic synthesis for relaxed memory models. In PLDI,
pages 429–440, New York, NY, USA, 2012.
[29] Gary L. Peterson. Myths about the mutual exclusion problem. Inf.
Process. Lett., 12(3):115–116, 1981.
[30] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of
concurrent software. In TACAS, volume 3440 of LNCS, pages 93–107.
Springer, 2005.
[31] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality
constraints. In Principles and Practice of Constraint Programming (CP),
volume 3709 of LNCS, pages 827–831. Springer, 2005.
[32] B. K. Szymanski. A simple solution to Lamport’s concurrent program-
ming problem with linear wait. In ICS, pages 621–626, New York, NY,
USA, 1988.