Synthesis of Memory Fences via Refinement Propagation

Yuri Meshman[2], Andrei Dan[1], Martin Vechev[1], and Eran Yahav[2]

[1] ETH Zurich
{andrei.dan, martin.vechev}@inf.ethz.ch
[2] Technion
{yurime, yahave}@cs.technion.ac.il

Abstract. We address the problem of fence inference in infinite-state concurrent programs running on relaxed memory models such as TSO and PSO. We present a novel algorithm that can automatically synthesize the necessary fences for infinite-state programs.
Our technique is based on two main ideas: (i) *verification with numerical domains*: we reduce verification under relaxed models to verification under sequential consistency using integer and boolean variables. This enables us to combine abstraction refinement over booleans with powerful numerical abstractions over the integers. (ii) *synthesis with refinement propagation*: to synthesize fences for a program P, we combine abstraction refinements used for successful synthesis of programs coarser than P into a new candidate abstraction for P. This "proof reuse" approach dramatically reduces the time required to discover a proof for P.
We implemented our technique and successfully applied it to several challenging concurrent algorithms, including state of the art concurrent work-stealing queues.

1 Introduction

Modern architectures use relaxed memory models in which memory operations may be reordered and executed non-atomically [2]. To allow programmer control over those orderings, processors provide special *memory fence* instructions. Unfortunately, manually reasoning where to place fences in a concurrent program running on a relaxed architecture is a challenging task. Using too many fences hinders performance, while missing necessary fences leads to incorrect programs.

Placing Memory Fences Finding a correct and efficient fence assignment is important for expert designers of concurrent algorithms as well as for developers wishing to implement a concurrent algorithm from the literature (these algorithms are regularly published without any mention of fences). Yet, manually finding the right fence assignment is difficult as these algorithms often rely on subtle ordering of events, which may be violated under relaxed memory models [11, Ch.7]. Further, the process of placing fences has to be repeated whenever the algorithm changes or is ported to another architecture.

Our Approach In this work we propose a novel automatic framework for synthesis of memory fences that can handle infinite-state programs. Given a program P, a safety specification S, an abstraction α and a memory model M, our system automatically synthesizes a memory fence assignment f such that the program P with fence assignment f (denoted by $P\langle f\rangle$) can be shown to satisfy the specification S under M using α,

that is $[\![P\langle f\rangle]\!]^{\alpha}_{M} \models S$. This is a particularly challenging task as even automatic verification is a difficult problem: currently there is very little work on automatic verification of infinite-state concurrent programs [8, 1] running on relaxed architectures, yet most concurrent algorithms are infinite-state (e.g. [23, 6]). Our system is based on two key ideas.

Synthesis via Abstraction Refinement Across Programs First, we introduce a synthesis algorithm which explores the abstraction refinements needed to verify a program P by *combining* abstraction refinements used for successful verification of programs *coarser* than P (programs that use a superset of fences). This is important as finding an abstraction refinement that is precise enough to verify a concurrent program is known to be a difficult problem. Our "proof reuse" approach reduces the time required to prove P. To the best of our knowledge, this is the first work which performs abstraction refinement by learning information *across multiple programs*, as opposed to the traditional abstraction refinement typically performed within a single program.

Verification via Reduction with Numerical Abstract Domains Second, we verify a program under relaxed memory models by reduction to a program under sequential consistency. This reduction approach, also advocated by other works [3, 8], is powerful as it enables one to leverage advances in the analysis of concurrent programs under sequential consistency. Based on this general idea, we reduce the verification problem under relaxed models to a problem of verification under sequential consistency *using integer and boolean variables*.

This reduction enables us to use powerful numerical abstract domains such as Polyhedra [7] and allows us for the first time to verify properties of infinite state concurrent algorithms such as the Chase-Lev [6] and THE [9] work stealing queues. However, numerical domains are insufficient by themselves as they can only represent convex information and the non-determinism introduced by relaxed memory models often requires capturing *disjunctions*. To track such information precisely, we leverage the expressive power of an abstract domain that *combines numerical information with finite boolean information (predicates)*. We track the non-deterministic aspects of the relaxed memory model using disjunctions in the finite part of the domain.

Main Contributions The novel contributions of our system are:

- A verification procedure based on transforming a program under relaxed semantics into a program under sequential consistency, enabling application of powerful numerical abstract domains. To refine the abstraction, we show how to track the non-deterministic aspects (which induce non-convex information) inherent in relaxed memory models via disjunctions encoded in the finite part of the domain.
- An efficient synthesis procedure which searches for minimal fence assignments by combining abstraction refinements used in successful proofs of coarser programs.
- An implementation and evaluation of our system for the x86-TSO and PSO memory models instantiated with classical numerical domains such as Polyhedra. We performed an extensive experimental study on a set of 15 concurrent algorithms. We believe this is the first time classic abstract interpretation has been used to prove properties of infinite-state work-stealing queues [6, 9].

2 Overview

Thread 1:

```
1 flag0 = 1;
2 turn = 1;
3 f1 = flag1;
4 lt1 = turn;
5 if ((lt1 != 0) & (f1 != 0))
6   goto 3;
7 nop; // CS
8 flag0 = 0;
9 goto 1;
```

Thread 2:

```
1 flag1 = 1;
2 turn = 0;
3 f2 = flag0;
4 lt2 = turn;
5 if ((f2 != 0) & (lt2 = 0))
6   goto 3;
7 nop; // CS
8 flag1 = 0;
9 goto 1;
```

assert always $((pc1 \neq 7) \vee (pc2 \neq 7))$

Fig. 1. Peterson mutual exclusion algorithm

In this section, we provide an informal overview of our approach using Peterson's mutual exclusion algorithm (shown in Fig. 1). More elaborate examples are considered in Section 5.

2.1 Motivating Example

In Fig. 1, each of the two threads attempts to reach their critical section (CS) at Line 7. To enter the critical section, a thread first checks whether the other thread intends to enter the critical section (by checking the value of `flag0` or `flag1`), as well as its turn (value of `turn`).

Our goal is to guarantee that both threads do not enter the critical section simultaneously. This property holds when the two threads run on a sequentially consistent machine, but no longer holds when they run on a relaxed memory model such as PSO or TSO. Under relaxed memory models, the writes to `flag0`, `flag1`, and `turn` performed by one thread may be buffered and not yet visible to the other thread when it reaches the condition at Line 5. As a result, both processes may enter the critical section simultaneously. For example, if thread 1 enters the critical section, and its write to `flag0=1` has not yet been flushed to main memory, thread 2 will pass its check at Line 5 and also enter the critical section. To guarantee that the mutual exclusion holds under relaxed memory models, the programmer has to explicitly add *memory fences* to the program. However, because fences are expensive, the programmer faces the challenge of inserting the minimal set of sufficient fences that makes mutual exclusion hold.

2.2 Searching for fence assignment and refinement placement

Our goal is to synthesize a minimal *fence assignment* for a given program, specification, and memory model. Finding such a minimal fence assignment involves a search over the space of possible fences and automatically checking the correctness of each program in the space. To automatically verify a program, we employ *abstraction refinement*. In our setting, abstraction refinement is described as a set of program locations (discussed

in detail later) which we refer to as a *refinement placement*.This leads to the following two-dimensional synthesis challenge:

Find a refinement placement and a minimal fence assignment which verify the program

Naive approach A naive approach where we perform an exhaustive search of the fence/refinement space is almost always non-feasible. For example, even for Peterson's algorithm, there are 2^6 potential fence assignments and 2^{23} potential refinement placements (we explain these in Section 2.3), leading to a total number of 2^{29} points in the fence/refinement space!

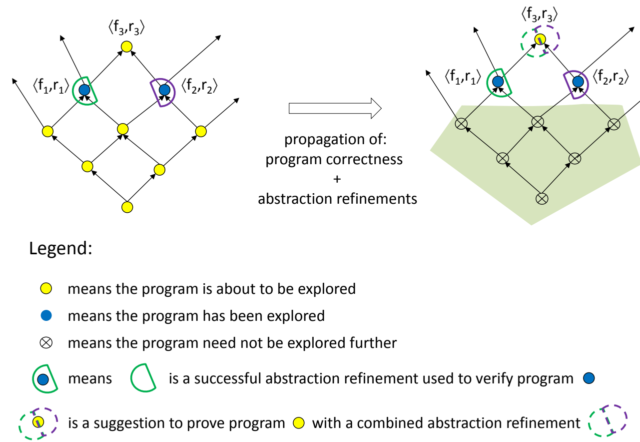


Fig. 2. Propagation of program correctness and abstraction refinements.

Our approach: semantic program and proof propagation Our approach works by pruning large parts of the search space, based on the following two observations (here we use the notation $P\langle f, r\rangle$ to mean program P with fence assignment f and refinement placement r):

- *implied correctness:* if the program $P\langle f, r\rangle$ is verified successfully, then it *implies the correctness* of any other point in space which uses a superset of the fences in f or a superset of the refinement locations in r.
- *implied incorrectness:* if the program $P\langle f, r\rangle$ fails to verify, then it *implies the incorrectness* of any other point in space which uses a subset of the fences in f or a subset of the refinement locations in r.

Fig. 2 shows an example of one of our propagation techniques (discussed in Section 3.1) and is meant to give an intuition. Here, successful verification of $P\langle f_1, r_1\rangle$ and $P\langle f_2, r_2\rangle$ implies the correctness of all programs in the search space "below" these two

(all programs with a subset of the fences). Further exploration of the space can first attempt to verify the point $P\langle f_3, r_3\rangle$ which employs a smaller set of fences ($f_3 = f_1 \cap f_2$) yet uses a refinement placement which *combines successful refinement placements from different programs* (i.e. r_1 and r_2). The intuition behind this combination is that slight relaxation of the program via fewer fences should only require slight adjustment of the abstraction refinement. Our experimental evaluation (Section 5) shows that propagation is effective for finding a minimal fence assignment for many of our benchmarks.

2.3 Refinement Placement - Reduction & Abstraction

We next describe several ingredients of our approach to verification of infinite-state programs running on relaxed memory models.

As described in the motivating example, on a relaxed memory model, writes to shared memory are not immediately visible to all processes: writes are first placed into a local buffer and then (non-deterministically), a flush instruction pops values from that buffer and writes them to shared memory. In our setting, this mechanism is encoded in source code via a translation phase.

Non-determinism due to flushes Fig. 3 shows the translation of the statement `flag0=1` for the PSO memory model (in this model, each thread maintains a FIFO buffer for each shared variable). Intuitively, `flag0=1` is translated into two parts: i) the write to the FIFO buffer and a non-deterministic `flush`. Details of this translation are discussed in Section 4. Here, we only discuss the translation of the `flush`. A `flush` from a store buffer works by writing back to shared memory an arbitrary number of items from the buffer. This is captured by the `while (*)` loop that has a non-deterministic termination condition (denoted by $*$).

```
/* begin store */
if flag0_cnt_0 > 0 {
    overflow = true;
    halt;
}
flag0_cnt_0 = flag0_cnt_0 + 1;
if flag0_cnt_0 < 2
    flag0_1_0 = 1;
/* end store */
/* begin flush */
yield;
while * do {
    if flag0_cnt_0 > 0 {
        flag0 = flag0_1_0;
        flag0_cnt_0 = flag0_cnt_0 - 1;
        yield;
    }
}
/* end flush */
yield;
```

Fig. 3. Translation of `flag0=1` under PSO.

The non-deterministic loop introduces a significant challenge when reasoning with numerical domains (which capture state via relations between variables). The reason is that two program states appearing right after the while loop has completed differ significantly depending on whether the `flush` was performed or not. Both states can be captured with disjunctions, but standard (convex) numerical domains often dramatically lose precision exactly in such (disjunctive) cases.

Local abstraction refinement To address this loss of precision, we use an abstract domain that combines numerical information with a finite boolean domain. By carefully

introducing boolean predicates, we can refine the abstraction (by splitting the numerical state) in a *local* manner. While local refinement may restore sufficient precision for successful verification, it unfortunately comes at an exponential cost. The addition of new predicates can lead to an exponential blowup of the program analysis, as each predicate may double the state space. Further, such a refinement is not required for all locations of a `flush`. For example, if we can prove that a `flush` is always reached with an empty buffer, the `flush` will have no effect, and thus there is no need to refine the abstraction at such locations (we elaborate on this point in Section 4.4).

This introduces the challenge of finding a suitable *refinement placement* (a subset of the `flush` program locations) that is precise enough to enable verification yet is scalable enough for the analysis to terminate in reasonable time.

3 Abstraction-Guided Fence Synthesis

In this section, we present a new synthesis algorithm which propagates both fence assignments and refinement placements. Our algorithm leverages implied correctness/incorrectness to reduce the search space. The algorithm treats the two dimensions of the problem as having the same importance, and looks for a minimal fence assignment and a minimal refinement. In addition, the algorithm strives to minimize the number of fences based on a new concept where an abstraction refinement is obtained by combining successful refinements *across programs*.

3.1 Abstraction-Guided Fence Synthesis

Algorithm 1 provides a declarative description of our approach. The algorithm takes as input a program P, a specification S, a memory model M and an abstraction α, and produces a (possibly modified) program P' that satisfies the specification under M with a minimal verifiable fence assignment. The algorithm leverages information *from several points in the space* in the verification effort of a given point.

Fence assignment and refinement placement A *fence assignment* f for a program P with program labels Lab_p is simply a subset of program labels $f \subseteq Lab_p$. A *refinement placement* r for a program P is also a subset of program labels, but it is restricted only to program labels of flush operations. The details of the refinement are elaborated in Section 4 and are not important for understanding the central concept of the synthesis algorithm presented in this section. For a given fence assignment $\mathtt{f}$ and a refinement placement $\mathtt{r}$, $P\langle \mathtt{f}, \mathtt{r} \rangle$ denotes the program P with fences placed according to $\mathtt{f}$ and an abstraction refinement selected according to $\mathtt{r}$.

Searching for satisfying placements The algorithm begins by initializing a `worklist` with (i) the program under a full fence assignment (Line 3) together with (ii) a refinement placement of program locations that are reasonable (Line 4) as determined by our Empty Buffer Analysis (EBA) (see Section 4.4).

For each element of the `worklist`, the algorithm tries to improve the fence assignment and refinement placement (Lines 8 and 9). The operation of these two functions is discussed later in this section. Our algorithm then invokes the underlying verifier to check if $[\![P\langle \mathtt{f}, \mathtt{r} \rangle]\!]^{\alpha}_{M} \models [\![S]\!]^{\alpha}_{M}$ (Line 10).

Optimized Semantic Search Our algorithm maintains the two sets `verified` and `falsified` for storing points $\langle \mathtt{f}, \mathtt{r} \rangle$ that have been verified or where verification failed, respec-

Input: P - program, S - Spec, M - memory model, α - abstraction, s.t. $[\![P]\!]^{\alpha}_{SC} \models [\![S]\!]^{\alpha}$
Output: P' - program such that $[\![P']\!]^{\alpha}_{M} \models [\![S]\!]^{\alpha}_{M}$ with minimal a number of fences

```
1  verified = ∅
2  falsified = ∅
3  f = fullFenceAssignment(P)
4  worklist = {⟨f,EBA(P,f)⟩}
5  while worklist ≠ ∅ do
6  |  ⟨f,r⟩ = select some pair from worklist
7  |  known = verified ∪ falsified
8  |  f = improveF(f,known)
9  |  r = improveR(f,r,known)
10 |  if [[P⟨f,r⟩]]^α_M ⊨ [[S]]^α_M then
11 |  |  verified ∪= {⟨f̂,r̂⟩ | f ⊆ f̂, r ⊆ r̂}
12 |  |  alternatives = relax(f,r,known)
13 |  else
14 |  |  falsified ∪= {⟨f',r'⟩ | f' ⊆ f, r' ⊆ r}
15 |  |  alternatives = restrict(f,r,known)
16 |  end
17 |  worklist = (worklist ∪ alternatives) \ known
18 end
19 ⟨f,r⟩ = min(verified)
20 return P⟨f,r⟩
```

Algorithm 1: Semantic search for finding minimal verifiable fence assignments.

tively. Initially, both of these sets are empty. In the case of successful verification, the algorithm adds $\langle \texttt{f}, \texttt{r}\rangle$ to the set of verified points in space. However, the algorithm does more than that: it also adds to `verified` all points which consist of a superset of fences as well as a superset of refinements. Successful verification of $P\langle \texttt{f}, \texttt{r}\rangle$ means that the search can proceed to explore *more relaxed* versions of the program. The helper function `relax(f,r,K)` is used to compute a set of $\langle \texttt{f'}, \texttt{r'}\rangle$ pairs that admit more behaviors (via a subset of fences) as well as coarser abstractions:

$$\texttt{relax(f,r,K)} = \{\langle \texttt{f'}, \texttt{r'}\rangle \mid \texttt{f'} \subset \texttt{f} \text{ and } \texttt{r'} \subseteq \texttt{r} \text{ and } \langle \texttt{f'}, \texttt{r'}\rangle \notin \texttt{K}\}$$

In the case of failed verification, the algorithms can add $\langle \texttt{f}, \texttt{r}\rangle$ to the set of falsified points in space, but once again, it can do more than that. That is, the algorithm adds to the set `falsified` all points in the space which consist of a subset of fences and a subset of abstraction refinements. Failed verification $\langle \texttt{f}, \texttt{r}\rangle$ means that the search should explore *more restricted* versions of the program. The helper function `restrict(f,r,K)` computes a set of $\langle \texttt{f'}, \texttt{r'}\rangle$ pairs that admit fewer behaviors (via a superset of fences) as well as more refined abstractions:

$$\texttt{restrict(f,r,K)} = \{\langle \texttt{f'}, \texttt{r'}\rangle \mid \texttt{f} \subseteq \texttt{f'} \text{ and } \texttt{r} \subset \texttt{r'} \text{ and } \langle \texttt{f'}, \texttt{r'}\rangle \notin \texttt{K}\}$$

The algorithm terminates when there are no more alternatives to explore, and returns a program with a minimal fence assignment (in our implementation, we return all non-comparable minimal fence assignments).

Parametric choices Our algorithm is parameterized on three dimensions:

– the choice for the next pair $\langle$f, r$\rangle$ to select at Line 6. The method of choosing the next element determines if our search will be similar to a depth-first search, to a breadth-first search or to a search that explores random elements of the space.
– the function `ImproveF`. This function leverages the knowledge of previous verification attempts (i.e., from the set `known`). For example, if `f` already verified for a different refinement `r'` and `f'` is a configuration in `known` which verified, then we can inspect `f ∩ f'` instead of `f`.
– the function `ImproveR`. With this function we improve the refinement `r` based on available knowledge (i.e., from the set `known`). For a fence assignment `f` and a refinement `r`, if $\langle$f', r$\rangle$ and $\langle$f'', r'$\rangle$ both previously successfully verified and if `f'` and `f''` are stronger than `f` (that is, a superset of fences), then `ImproveR` will return `r ∪ r'`. Intuitively, this makes the abstraction refinement more precise, increasing the chances of success.

4 Automatic Verification

In this section we discuss the three steps of our automatic verification procedure: the reduction procedure, the underlying program analysis and the mechanism of abstraction refinement. We also discuss a static *empty buffer analysis* that is used by our algorithm to compute a set of possible abstraction placements to chose from (as discussed earlier).

4.1 Reduction

Similarly to [3, 8], we reduce a program P running on a relaxed model M to a program P_M running on sequential consistency. This enables us to directly leverage advances in program analysis for sequential consistency. We adopt a similar translation procedure to [8] where the key idea in constructing P_M is representing the abstraction of the store buffers of M as variables in P_M. We illustrate the process for when M is the PSO memory model. The process for x86-TSO is similar. For PSO, it is sufficient to consider a program P_{PSO} where every shared variable X in the program P is also associated with: (i) additional k *local* variables for each thread t: $x_{1_t}, \ldots, x_{k_t}$, representing the content of a local store buffer for this variable in each thread t, (ii) a buffer counter variable x_{cnt_t} that records the current position in the store buffer of X in thread t.

The translation uses the function $[\![]\!]$ which takes as input a statement S, a thread t, and a bound k on the maximum buffer size and produces a new statement as output $[\![\mathrm{S}]\!]^t_k$ (Fig. 4). The translation procedure is described in detail in [8]. Let us take a closer look at the most challenging method, the `flush`.

A `flush` is translated into a non-deterministic loop. If the buffer counter for the variable is positive, then it non-deterministically decides whether to update the shared variable X. If it has decided to update X, the earliest write (i.e. x_{1_t}) is stored in X. The contents of the local variables are then updated by shifting: the content of each x_{i_t} is taken from the content of the successor $x_{(i+1)_t}$ where $1 \leq i < k$. Finally, the buffer count is decremented.

In our encoding of concurrent programs, context switches between threads are explicitly specified with `yield` statements (we place `yield` statements after every instruction). Because under the relaxed memory model a `flush` can be executed non-deterministically by the memory subsystem at any moment during program execution, our reduction places a (translated) `flush` after every `yield`.

$[\![X = r]\!]_k^t$

if $x_{cnt_t} = k$ **then**
abort("overflow");
$x_{cnt_t} = x_{cnt_t} + 1$;
if $x_{cnt_t} = 1$ **then** $x_{1_t} = r$;
...
if $x_{cnt_t} = k$ **then** $x_{k_t} = r$;

$[\![r = X]\!]_k^t$

if $x_{cnt_t} = 0$ **then** $r = X$;
...
if $x_{cnt_t} = k$ **then** $r = x_{k_t}$;

$[\![\text{fence}]\!]_k^t$

▷ for each shared variable X generate:
assume $(x_{cnt_t} = 0)$;
▷ end of generation

$[\![\text{flush}]\!]_k^t$

while * **do**
  ▷ for each shared variable X generate:
  if $x_{cnt_t} > 0$ **then**
    if * **then**
      $X = x_{1_t}$;
      if $x_{cnt_t} > 1$ **then**
        $x_{1_t} = x_{2_t}$;
        ...
        if $x_{cnt_t} = k$ **then** $x_{(k-1)_t} = x_{k_t}$;
      end
      $x_{cnt_t} = x_{cnt_t} - 1$;
    end
  end
  ▷ end of generation
end

Fig. 4. PSO Translation Rules: each sequence is atomic

4.2 Analysis with Numerical Abstract Domains

Once we have obtained the reduced program P_M, we use abstract interpretation with advanced numerical domains to verify its properties under sequential consistency. In particular, if the property we are interested in verifying relates only to shared numerical variables G appearing in program P (for example, the property of no array access out of bounds), then when translating accesses to variables of G by P, the reduction to program P_M will only introduce additional numerical variables over the variables in G: these are the local variables and counters. This enables us to directly use powerful numerical domains such as the Polyhedra abstract domain over the resulting program P_M. There are three possible outcomes of the automatic verification step:

- The program P_M verifies in which case the verification is successful.
- The program P_M does not verify because an overflow occurred during the analysis. There could be two reasons why overflow occurs:
  - there exists a concrete execution in the program which indeed does lead to an overflow (e.g. multiple stores to a shared variable without a fence in between).
  - the abstraction is imprecise enough to establish that there is no overflow.

  We cannot distinguish between these two cases and hence, when overflow occurs, we increase k to a small bound (at most number of removed fences) or refine the abstraction (detailed below). Our experience is that small values of k combined with an abstraction refinement of the numerical analysis work well in practice.
- The program P_M does not verify because the property being checked fails to verify under the current abstraction. In this case, we typically apply abstraction refinement to the numerical analysis.

4.3 Abstraction Refinement of Numerical Analysis

As discussed above, abstraction refinement is often a vital step to enable successful verification of the program P_M. A key question then is which parts of the program P_M require a more refined treatment in the abstract? To find these statements in P_M, we employ a two-step approach, where we always first verify the program P under sequential consistency, before trying to verify the translated program P_M. This allows us to focus the search for abstraction refinement on the statements in P_M that are the root cause for the new behaviors. In our setting, these are the `flush` instructions appearing in P_M as it is via these statements that relaxed memory effects eventually become visible.

Abstraction refinement of the numerical analysis is accomplished in our system by directly encoding the suggested refinement into the program P_M by automatically introducing boolean auxiliary variables at places where the memory model relaxation takes effect. In particular, the number of boolean variables is proportional to k and these boolean variables are initialized appropriately inside the branches of the translated `flush` statement (to $true$ or $false$ respectively, depending at which branch the boolean variable is assigned). E.g. for peterson's algorithm (Fig. 1), under minimal verifiable fence placement for TSO, our analysis found that a boolean variable was needed at the flush after Thread 1 assignment to `turn` but not after that assignment for Thread 0.

This is yet another advantage of the reduction approach: it enables us to quickly experiment with and provide the abstraction refinements over the base numerical domains by modifying the program P_M instead of trying to somehow change the internals of an existing program analyzer (or build a new analysis). In particular, our system integrates with CIP [13] (which supports logico-numerical domains) enabling us to match the auxiliary boolean variables with the logical part of the combined domain.

Overall, we believe that we have found a good match between the particular type of abstraction refinement required in our context, the fact that this refinement can be encoded in the program and the ability of an existing analyzer to consume this encoding directly into its abstract domain.

4.4 Empty-Buffer Analysis

The additional predicates from refinement placement track the non-determinism due to flushes. However, such non-determinism is only relevant when the store buffers are not empty. When the store buffers are empty, a `flush` operation has no effect, and thus there is no need for a refinement at that program point. The challenge of course is to statically identify program locations in which the store buffers are guaranteed to be empty in any possible execution of the program. Towards that end, we use a simple static analysis that identifies program points where buffers are guaranteed to be empty. The analysis is sound, when it reports that a store buffer is empty, it guarantees that it will be empty in any possible execution. In Section 5, we show that the empty buffer analysis is effective and produces an upper bound on refinements that is significantly lower than the total number of possible locations.

5 Evaluation

We implemented our approach as described in previous sections and evaluated it on a range of challenging concurrent algorithms. To the best of our knowledge, this is the first extensive analysis study in the context of relaxed memory models involving

infinite-state reasoning and abstract interpretation. All of our experiments were performed on an Intel(R) Xeon(R) 2.13GHz machine with 250 GB RAM.

For the automatic verification step, we used ConcurInterProc [14] which uses the APRON numerical abstract domain library [15]. To check that the inferred invariants imply the specification, we used the Z3 SMT solver.

5.1 Concurrent Algorithms

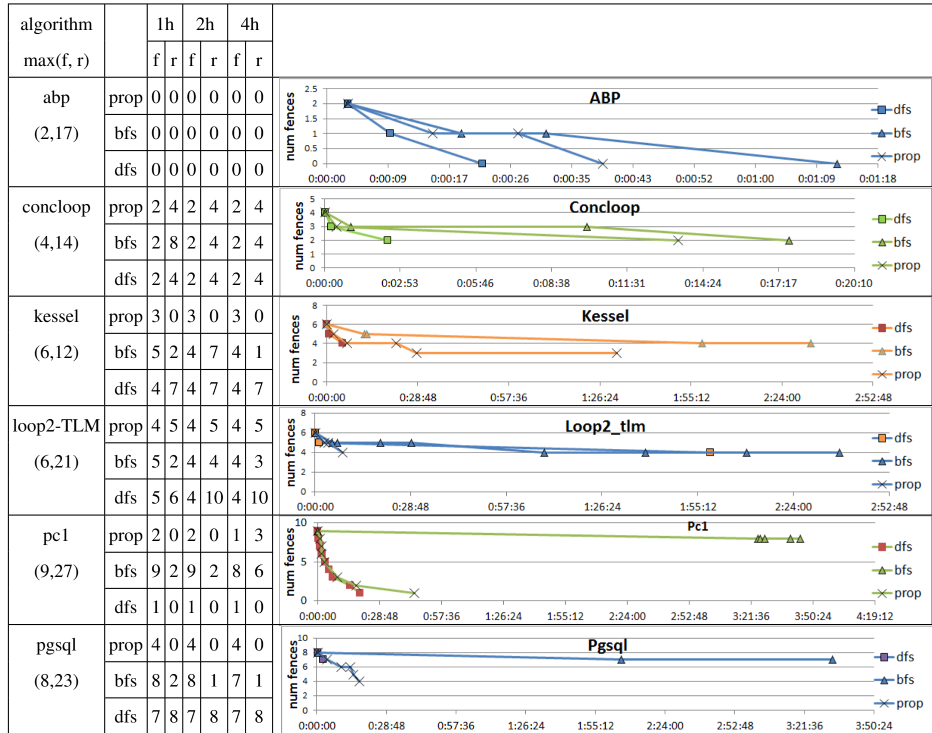
In our experiments we used 15 concurrent algorithms (7 finite-state and 8 infinite-state). Among these, there are 3 (infinite-state) array-based work-stealing queues and 7 mutual exclusion algorithms. We are not aware of any previous attempts to automatically verify properties of concurrent data structures such as the work stealing queues (WSQs) under relaxed models. For all of the algorithms we verified safety properties (e.g. a pair of labels is unreachable). For the WSQs, we verified consistency properties such as: the head index of the queue is always less than the tail index.

While our technique never reports incorrect fence assignments, due to non-monotonic analysis, the final result might lose the minimality guarantee. We note that in our benchmarks, this situation was never encountered. To cope with non-monotonicity, the tool has to spend more time searching when intermediate points fail to verify. In specific situations (certain outputs from ConcurInterProc), the search continues or even retries to verify a program when the verification tool ConcurInterProc returns “unknown”.

5.2 Results

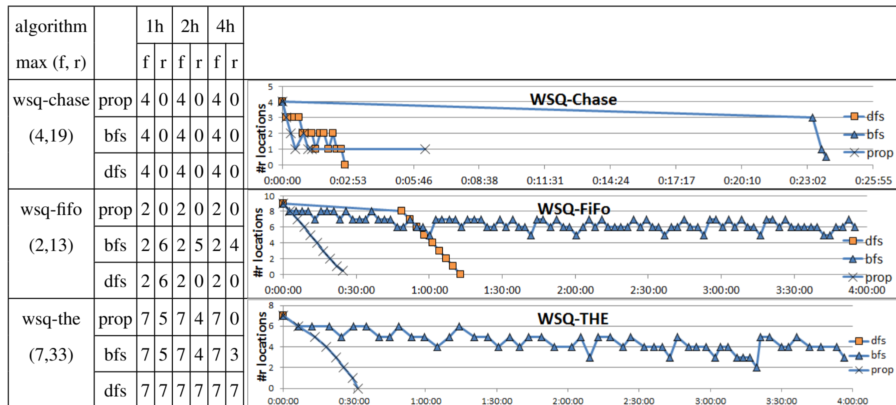
Our experimental results for both PSO and TSO memory models are summarized in Tables 1, 2, 3, and 4. Not all of the algorithms are shown due to space restriction. Graphs for the remaining results can be found in [22]. The first column of each table contains a tuple, under each benchmark name – the first element is the maximal number of fences for the algorithm, and the second element is the total number of locations for abstraction placements. For each benchmark, we bounded the search time to an hour, two hours and four hours. Each time bound result has two parts: the minimal number of fences achieved (columns labeled f) and the minimal relaxation under that fence assignment that the algorithm was able to find (columns labeled r). We compared three versions of our search: (i) breadth-first search (lines labeled bfs), (ii) depth-first search (lines labeled dfs) both without propagation and (iii) search with propagation (`prop`). At each point, the algorithm explores the next element from the worklist which is highest in the lattice for bfs, or lowest for dfs. After successful or failed verification and updating the set `known`, we update the worklist with the immediate successors of the attempted configuration (above or below the explored element - depending on whether it was verified or not). The third search configuration (labeled `prop`) is a bfs search *with propagation*.

The graphs depict the time it took to discover the minimal fence assignment. The x-axis is the time in a “hour:minute:seconds” format and the y-axis is the number of fences discovered. For some cases, such as PC1, it can be seen that the initial behavior of the `prop` approach is similar to that of dfs. This is due to a “streak” of successful verifications where a successful verification from a previous stage (say fence assignments “remove #9” and “remove #8” verified) affects the next element attempted by the `prop` approach (for the example given “remove #8 and #9” will be attempted). This behavior



algorithm max(f, r)		1h f	1h r	2h f	2h r	4h f	4h r
abp (2,17)	prop	0	0	0	0	0	0
	bfs	0	0	0	0	0	0
	dfs	0	0	0	0	0	0
concloop (4,14)	prop	2	4	2	4	2	4
	bfs	2	8	2	4	2	4
	dfs	2	4	2	4	2	4
kessel (6,12)	prop	3	0	3	0	3	0
	bfs	5	2	4	7	4	1
	dfs	4	7	4	7	4	7
loop2-TLM (6,21)	prop	4	5	4	5	4	5
	bfs	5	2	4	4	4	3
	dfs	5	6	4	10	4	10
pc1 (9,27)	prop	2	0	2	0	1	3
	bfs	9	2	9	2	8	6
	dfs	1	0	1	0	1	0
pgsql (8,23)	prop	4	0	4	0	4	0
	bfs	8	2	8	1	7	1
	dfs	7	8	7	8	7	8

Table 1. PSO results. The graphs show discovered fence assignments over time.



algorithm max (f, r)		1h f	1h r	2h f	2h r	4h f	4h r
wsq-chase (4,19)	prop	4	0	4	0	4	0
	bfs	4	0	4	0	4	0
	dfs	4	0	4	0	4	0
wsq-fifo (2,13)	prop	2	0	2	0	2	0
	bfs	2	6	2	5	2	4
	dfs	2	6	2	0	2	0
wsq-the (7,33)	prop	7	5	7	4	7	0
	bfs	7	5	7	4	7	3
	dfs	7	7	7	7	7	7

Table 2. PSO results. The graphs show discovered refinement placements over time.

algorithm (f, r)		1h f	1h r	2h f	2h r	4h f	4h r
abp (2,17)	prop	0	0	0	0	0	0
	bfs	0	0	0	0	0	0
	dfs	0	0	0	0	0	0
concloop (4,14)	prop	2	4	2	4	2	4
	bfs	2	8	2	4	2	4
	dfs	2	4	2	4	2	4
kessel (6,12)	prop	5	3	4	0	4	0
	bfs	5	3	5	1	5	1
	dfs	5	6	5	6	5	6
loop2-TLM (6,21)	prop	6	2	5	8	4	14
	bfs	5	8	5	8	5	7
	dfs	5	10	5	10	5	10
pc1 (9,27)	prop	3	3	3	3	1	14
	bfs	9	3	9	2	8	6
	dfs	5	9	5	9	5	9
peterson (6,23)	prop	5	2	4	3	4	3
	bfs	6	0	5	5	5	2
	dfs	4	7	4	7	4	7
pgsql (8,23)	prop	5	7	5	7	5	7
	bfs	8	4	8	3	8	1
	dfs	8	8	8	8	8	8

Table 3. TSO results. The graphs show discovered fence assignments over time

is similar to dfs. For algorithms such as KESSEL and PGSQL, it can be seen that the dfs approach finds early in the search a non-optimal fence assignment (the `prop` approach finds a better assignment later) and no new points appear in the graph. In those cases the dfs approach proceeds to explore lower elements in the lattice and fails repeatedly.

For several algorithms only the full assignment of fences was verified. Those algorithms are described in Table 2. Here (unlike Table 1), the graph's y-axis is the abstraction refinement placement that verified. Those graphs have more points and describe more clearly the difference between the three approaches (dfs, bfs and `prop`). In many cases, bfs explores "too many" elements high in the lattice, dfs converges fast to the lowest element it can verify but then it needs to backtrack. For WSQ-THE, dfs didn't find a placement smaller than the full one. Perhaps given more time it would "backtrack" and find a placement equivalent to the one the `prop` approach found.

Summary It can be seen that the search with propagation (`prop`) finds smaller fence assignments quicker than bfs and fewer or equal fence assignments than dfs.

algorithm (f, r)		1h		2h		4h		
		f	r	f	r	f	r	
queue (1,13)	prop	1	0	1	0	1	0	Queue TSO
	bfs	1	0	1	0	1	0	
	dfs	1	0	1	0	1	0	
wsq-chase (4,19)	prop	4	0	4	0	4	0	WSQ-Chase TSO
	bfs	4	0	4	0	4	0	
	dfs	4	0	4	0	4	0	

Table 4. TSO results - The graphs show discovered refinement placements over time.

6 Related Work

Next, we discuss some of the work that is most closely related to ours. These works include automatic verification (most closely related) techniques, dynamic analysis and bounded model checking approaches, search propagation in synthesis as well as robustness. Generally, while there has been some work on bounded model checking of concurrent programs running on relaxed memory models, there has been almost no work on automatically verifying *infinite-state* concurrent programs running on these models.

Program Transformation One general direction for handling relaxed memory model programs is to encode their effects into a program and then analyze the resulting program using standard tools geared towards sequential consistency. Towards that, the works of [3, 4] suggest source-to-source transformations which encode the relaxed memory semantics into the target program. We also believe that this is a viable path and in our work, we also use a similar encoding approach. However, as we have seen, direct encoding of the semantics is typically not sufficient when dealing with infinite-state programs where the precision of the abstraction is critical.

Handling infinite-state programs Kuperstein et al. [18] handle some forms of infiniteness (such as that coming from the buffers), but do not handle general infinite-state programs under sequential consistency. Other works in this direction are those of Linden et al. [19, 20] which shows how to use automata as symbolic representation of store buffers. Their work focused on programs that are finite-state under sequential consistency. The work of Vafeiadis et al. [25] presents an approach for eliminating fences under x86-TSO. Their approach is based on compiler transformations and assumes that the input program is correct. The work of Abdulla et al. [1] builds on [18] and is able to handle infinite-state programs under x86-TSO. That work combines predicate abstraction with the store buffers abstraction from [18]. The approach uses traditional abstraction refinement in order to discover the necessary predicates. Our recent work [8] handles both x86-TSO and PSO memory models and also uses predicate abstraction. However, the procedure for inferring the predicates necessary to verify the program under relaxed memory models differs from standard abstract refinement. Instead, the paper proposes a form of proof extrapolation: it first assumes that the program is verified under sequential consistency and then shows how to adapt these predicates (in a memory-model specific way) into new predicates which are then used as candidates for the verification under

the particular relaxed memory model. Both of these approaches are based on predicate abstraction and require the predicates to be inferred via refinement or adaptation. In contrast, the techniques presented in this work are based on iterative numerical abstract interpretation which promises to scale better (but is focused on numerical domains). In addition, our search algorithm combines propagation of abstraction refinements *across* programs with program restriction via fence inference. Our work also has relevance to the well known technique of lazy abstraction [10] which introduces the concept of adjusting the level of abstraction for different sections of the verified program's state space. In our approach, the search can be seen as selectively introducing refinements which guide the analyzer. However, unlike previous work, we learn new refinements by combining existing successful refinements from *several* programs.

Explicit Model Checking for Relaxed Memory Models There have been several works (e.g., [18, 16, 17, 12]) focusing on explicit-state model checking under relaxed memory models. Among those, [17] focuses on fence inference and [12] also describes an explicit-state model checking and inference technique for the .NET memory model. These approaches are sound only for finite-state programs, and cannot handle infinite-state programs. CheckFence [5] takes a different approach, instead of working with operational memory models and explicit model-checking, they convert a program into a form that can be checked against an axiomatic model specification. This technique unrolls loops at a preprocessing stage and cannot handle infinite-state programs.

In addition, there has recently been interest in exploring dynamic techniques for testing programs running on various relaxed memory models. The work of Liu et al. [21] dynamically analyzes (via a demonic scheduler) concurrent algorithms under the TSO and PSO memory models and whenever it finds a violating trace proposes a repair which inserts memory fences into the program. Recently, there has also been work on leveraging various partial order reduction techniques for bounded model checking of concurrent C++ programs [24]. Both of these works attempt to handle larger programs by sacrificing soundness.

7 Conclusion

In this work, we presented a system that can automatically synthesize fences in infinite-state concurrent algorithms running on relaxed memory models such as TSO and PSO. Our system is based on two core ideas.

First, in addition to propagating correctness between different fence assignments, the synthesizer explores the space of programs by using a form of "proof propagation": computing a candidate abstraction refinement of a given program by combining successful abstraction refinements of coarser programs. Second, we reduce the problem of automatic verification under a relaxed memory model into one of verification under sequential consistency using only integer and boolean variables. This enables us to leverage powerful numerical abstractions over the integers and to refine these abstractions by directly encoding the boolean refinement in the reduced program.

Finally, we evaluated our system on 15 challenging concurrent algorithms, including concurrent work-stealing queues. We believe that this is the first extensive study of using abstract interpretation techniques in the context of relaxed memory models and the first time properties of some of these algorithms have been verified.

References

1. ABDULLA, P. A., ATIG, M. F., CHEN, Y.-F., LEONARDSSON, C., AND REZINE, A. Automatic fence insertion in integer programs via predicate abstraction. SAS'12.
2. ADVE, S. V., AND GHARACHORLOO, K. Shared memory consistency models: A tutorial. *IEEE Computer 29* (1995), 66–76.
3. ALGLAVE, J., KROENING, D., NIMAL, V., AND TAUTSCHNIG, M. Software verification for weak memory via program transformation. ESOP'13.
4. ATIG, M. F., BOUAJJANI, A., AND PARLATO, G. Getting rid of store-buffers in tso analysis. In *CAV'11* (2011).
5. BURCKHARDT, S., ALUR, R., AND MARTIN, M. M. K. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI '07* (2007).
6. CHASE, D., AND LEV, Y. Dynamic circular work-stealing deque. In *SPAA* (2005).
7. COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among variables of a program. In *POPL'78* (1978).
8. DAN, A. M., MESHMAN, Y., VECHEV, M. T., AND YAHAV, E. Predicate abstraction for relaxed memory models. In *SAS'13* (2013).
9. FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. PLDI '98.
10. HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Lazy abstraction. POPL '02.
11. HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Apr. 2008.
12. HUYNH, T. Q., AND ROYCHOUDHURY, A. Memory model sensitive bytecode verification. *Form. Methods Syst. Des. 31*, 3 (Dec. 2007).
13. JEANNET, B. The CONCURINTERPROC interprocedural analyzer for concurrent programs. `http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi`.
14. JEANNET, B. Relational interprocedural verification of concurrent programs. *Software and System Modeling 12*, 2 (2013), 285–306.
15. JEANNET, B., AND MINÉ, A. Apron: A library of numerical abstract domains for static analysis. In *CAV* (2009), A. Bouajjani and O. Maler, Eds., vol. 5643, pp. 661–667.
16. JONSSON, B. State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). *SIGARCH Comput. Archit. News 36*, 5 (June 2009), 65–71.
17. KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Automatic inference of memory fences. In *FMCAD '10* (2010).
18. KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Partial-coherence abstractions for relaxed memory models. PLDI '11.
19. LINDEN, A., AND WOLPER, P. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN* (2010), pp. 212–226.
20. LINDEN, A., AND WOLPER, P. A verification-based approach to memory fence insertion in pso memory systems. TACAS'13.
21. LIU, F., NEDEV, N., PRISADNIKOV, N., VECHEV, M., AND YAHAV, E. Dynamic synthesis for relaxed memory models. PLDI '12.
22. MESHMAN, Y., DAN, A., VECHEV M., AND YAHAV, E. Synthesis of memory fences via refinement propagation. Tech. rep.
23. MICHAEL, M. M., VECHEV, M. T., AND SARASWAT, V. A. Idempotent work stealing. PPoPP '09.
24. NORRIS, B., AND DEMSKY, B. CDSchecker: checking concurrent data structures written with c/c++ atomics. In *OOPSLA '13* (2013), OOPSLA '13.
25. VAFEIADIS, V., AND NARDELLI, F. Z. Verifying fence elimination optimisations. In *SAS'11*.