Automatic Inference of Memory Fences

Michael Kuperstein
Technion

Martin Vechev
IBM Research

Eran Yahav
IBM Research and Technion,
Deloro Fellow

***Abstract*—This paper addresses the problem of placing memory fences in a concurrent program running on a relaxed memory model. Modern architectures implement relaxed memory models which may reorder memory operations or execute them non-atomically. Special instructions called *memory fences* are provided to the programmer, allowing control of this behavior. To ensure correctness of many algorithms, in particular of non-blocking ones, a programmer is often required to explicitly insert memory fences into her program. However, she must use as few fences as possible, or the benefits of the relaxed architecture may be lost. Placing memory fences is challenging and very error prone, as it requires subtle reasoning about the underlying memory model.**

We present a framework for automatic inference of memory fences in concurrent programs, assisting the programmer in this complex task. Given a finite-state program, a safety specification and a description of the memory model, our framework computes a set of ordering constraints that guarantee the correctness of the program under the memory model. The computed constraints are maximally permissive: removing any constraint from the solution would permit an execution violating the specification. Our framework then realizes the computed constraints as additional fences in the input program.

We implemented our approach in a tool called FENDER and used it to infer correct and efficient placements of fences for several non-trivial algorithms, including practical concurrent data structures.

I. Introduction

On the one hand, memory barriers are expensive (100s of cycles, maybe more), and should be used only when necessary. On the other, synchronization bugs can be very difficult to track down, so memory barriers should be used liberally, rather than relying on complex platform-specific guarantees about limits to memory instruction reordering. – Herlihy and Shavit, The Art of Multiprocessor Programming [1].

Modern architectures use relaxed memory models in which memory operations may be reordered and executed non-atomically [2]. These models enable improved hardware performance with respect to the standard sequentially consistent model [3]. However, they pose a burden on the programmer, forcing her to reason about non-sequentially consistent program executions. To allow programmer control over those executions, processors provide special *memory fence* instructions.

As multicore processors become increasingly dominant, highly-concurrent algorithms emerge as critical components of many systems [4]. Highly-concurrent algorithms are notoriously hard to get right [5] and often rely on subtle ordering of events, an ordering that may be violated under relaxed memory models (cf. [1, Ch.7]).

Finding a *correct and efficient* placement of memory fences for a concurrent program is a challenging task. Using too many fences (over-fencing) hinders performance, while using too few fences (under-fencing) permits executions that violate correctness. Manually balancing between over- and under-fencing is very difficult, time-consuming and error-prone as it requires reasoning about non sequentially consistent executions (cf. [1], [6], [7]). Furthermore, the process of finding fences has to be repeated whenever the algorithm changes, and whenever it is ported to a different architecture.

Our Approach In this paper, we present a tool that automatically infers *correct and efficient* fence placements. Our inference algorithm is defined in a way that makes the dependencies on the underlying memory model explicit. This makes it possible to use our algorithm with various memory models. To demonstrate the applicability of our approach, we implement a relaxed memory model that supports key features of modern relaxed memory models. We use our tool to automatically infer fences for several state of the art concurrent algorithms, including popular lock-free data structures.

Main Contributions The main contributions of this paper are:

- A novel algorithm that automatically infers a correct and efficient placement of memory fences in concurrent programs.
- A prototype implementation of the algorithm in a tool capable of inferring fences under several memory models.
- An evaluation of our tool on several highly concurrent practical algorithms such as: concurrent sets, work-stealing queues and lock-free queues.

II. Existing Approaches

We are aware of two existing tools designed to assist programmers with the problem of finding a correct and efficient placement of memory fences. However, both of these suffer from significant drawbacks.

CheckFence In [7], Burckhardt et al. present "CheckFence", a tool that checks whether a specific fence placement is correct for a given program under a relaxed memory model. In terms of checking, "CheckFence" can only consider finite executions of a linear program and therefore requires loop unrolling. Code that utilizes spin loops requires custom manual reductions. This makes the tool unsuitable for checking fence placements in algorithms that have unbounded spinning (e.g. mutual exclusion and synchronization barriers). To use "CheckFence" for inference, the programmer uses an iterative process: she starts with an initial fence placement and if the placement is incorrect, she has to examine the (non-trivial) counterexample from the tool, understand the cause of error and attempt to fix it by placing a memory fence at some program location. It is also possible to use the tool by starting with a very conservative placement and choose fences to remove until a counterexample is encountered. This process, while simple, may easily lead to a "local minimum" and an inefficient placement.

mmchecker presented in [8] focuses on model-checking with relaxed memory models, and also proposes a naive approach for fence inference. Huynh et. al formulate the fence inference problem as a minimum cut on the reachability graph. While the result produced by solving for a minimum cut is sound, it is often suboptimal. The key problem stems from the lack of one-to-one correspondence between fences and removed edges. First, the insertion of a single fence has the potential effect of removing many edges from the graph. So it is possible that a cut produced by a single fence will be much larger in terms of edges than that produced by multiple fences. [8] attempts to compensate for this by using a weighing scheme, however this weighing does not provide the desired result. Worse yet, the algorithm assumes that there exists a *single* fence that can be used to remove any given edge. This assumption may cause a linear number of fences to be generated, when a single fence is sufficient.

III. Overview

In this section, we use a practically motivated scenario to illustrate why manual fence placement is inherently difficult. Then we informally explain our inference algorithm.

A. Motivating Example

Consider the problem of implementing the Chase-Lev work-stealing queue [9] on a relaxed memory model. Work stealing is a popular mechanism for efficient load-balancing used in runtime libraries for languages such as Java, Cilk and X10. Fig. 1 shows an implementation of this algorithm in C-like pseudo-code. For now we ignore the fences shown in the code.

The data structure maintains an expandable array of items called wsq and two indices top and $bottom$ that can wrap around the array. The queue has a single owner thread that can only invoke the operations `push()` and `take()` which operate on one end of the queue, while other threads call `steal()` to take items out from the opposite end. For simplicity, we assume that items in the array are integers and that memory is collected by a garbage collector (manual memory management presents orthogonal challenges [10]).

We would like to guarantee that there are no out of bounds array accesses, no lost items overwritten before being read, and no phantom items that are read after being removed. All these properties hold for the data structure under a sequentially consistent memory model. However, they may be violated when the algorithm executes on a relaxed model.

Under the SPARC RMO [11] memory model, some operations may be executed out of order. Tab. I shows possible reorderings under that model (when no fences are used) that lead to violation of the specification. The column *locations* lists the two lines in a given method which contain memory operations that might get reordered and lead to a violation. The next column gives an example of an undesired effect when the operations at the two labels are reordered. There could be other possible effects (e.g., program crashes), but we list only one. The last column shows the type of fence that can be used to prevent the undesirable reordering. Informally, the type describes what kinds of operations have to complete before other type of operations. For example, a store-load fence executed by a processor forces all stores issued by that processor to complete before any new loads by the same processor start.

```
1  typedef struct {
2    long size;
3    int  *ap;
4  } item_t;
5
6  long top, bottom;
7  item_t *wsq;
```

```
1  void push(int task) {
2   long b = bottom;
3   long t = top;
4   item_t* q = wsq;
5   if (b−t ≥ q→size −1){
6     q = expand();
7   }
8   q→ap[b % q→size]=task;
    fence("store−store");
9   bottom = b + 1;
10 }
```

```
1  int take() {
2   long b = bottom − 1;
3   item_t* q = wsq;
4   bottom = b;
    fence("store−load");
5   long t = top;
6   if (b < t) {
7     bottom = t;
8     return EMPTY;
9   }
10  task = q→ap[b % q→size];
11  if (b > t)
12    return task;
13  if (!CAS(&top, t, t+1))
14    return EMPTY;
15  bottom = t + 1;
16  return task;
17 }
```

```
1  int steal() {
2   long t = top;
    fence("load−load");
3   long b = bottom;
    fence("load−load");
4   item_t* q = wsq;
5   if (t ≥ b)
6     return EMPTY;
7   task=q→ap[t % q→size];
    fence("load−store");
8   if (!CAS(&top, t, t+1))
9     return ABORT;
10  return task;
11 }
```

```
1  item_t* expand() {
2    int  newsize = wsq→size * 2;
3    int* newitems = (int *) malloc(newsize*sizeof(int));
4    item_t *newq =  (item_t *)malloc(sizeof(item_t));
5    for (long i = top; i < bottom; i++) {
6      newitems[i % newsize] = wsq→ap[i % wsq→size];
7    }
8    newq→size = newsize;
9    newq→ap = newitems;
     fence("store−store");
10   wsq = newq;
11   return newq;
12 }
```

Fig. 1. Pseudo-code of the Chase-Lev work stealing queue [9].

#	Locations	Effect of Reorder	Needed Fence
1	push:8:9	`steal()` returns phantom item	store-store
2	take:4:5	lost items	store-load
3	steal:2:3	lost items	load-load
4	steal:3:4	array access out of bounds	load-load
5	steal:7:8	lost items	load-store
6	expand:9:10	`steal()` returns phantom item	store-store

TABLE I
Potential reorderings of operations in the Chase-Lev algorithm of Fig. 1 running on the RMO memory model.

Avoiding Failures with Manual Insertion of Fences To guarantee correctness under the RMO model, the programmer can try to manually insert fences that avoid undesirable reorderings. As an alternative to placing fences based on her intuition, the programmer can use an existing tool such as CheckFence [7] as described in Section II. Repeatedly adding fences to avoid each counterexample can easily lead to over-fencing: a fence used to fix a counterexample may be made redundant by another fence inferred for a later counterexample. In practice, localizing a failure to a single reordering is challenging and time consuming as a failure trace might include multiple reorderings. Furthermore, a single reordering can exhibit multiple failures, and it is sometimes hard to identify the cause underlying an observed failure. Even under the assumption that each failure has been localized to a single reordering (as in Tab. I), inserting fences still requires considering each of these 6 cases.

In a nutshell, the programmer is required to manually produce Tab. I: summarize and understand all counterexamples from a checking tool, localize the cause of failure to a single reordering, and propose a fix that eliminates the counterexample. Further, this process might have to be repeated manually every time the algorithm is modified or ported to a new memory model. For example, the fences shown in Fig. 1 are required for the RMO model, but on the SPARC TSO model the algorithm only requires the single fence in `take()`. Keeping all of the fences required for RMO may be inefficient for a stronger model, but finding which fences can be dropped might require a complete re-examination.

Automatic Inference of Fences It is easy to see that the process of manual inference does not scale. In this paper, we present an algorithm and a tool that automates this process. The results of applying our tool on a variety of concurrent algorithms, including the one in this section, are discussed in detail in Section V.

B. Description of the Inference Algorithm

Our inference algorithm works by taking as input a finite-state program, a safety specification and a description of the memory model, and computing a constraint formula that guarantees the correctness of the program under the memory model. The computed constraint formula is maximally permissive: removing any constraint from the solution would permit an execution violating the specification.

Applicability of the Inference Algorithm Our approach is applicable to any operational memory model on which we can define the notion of an *avoidable transition* that can be prevented by a *local* (per-processor) fence. Given a state, this requires the ability to identify: (i) that an event happens out of order; (ii) what alternative events could have been forced to happen instead by using a local fence. Requirement (i) is fairly standard and is available in common operational memory model semantics. Requirement (ii) states that a fence only affects the order in which instructions execute for the given processor but not the execution order of other processors. This holds for most common models, but not for PowerPC, where the SYNC instruction has a cumulative effect [12].

State Given a memory model and a program, we can build the transition system of the program, i.e. explore all reachable states of the program running on that memory model. A state in such a transition system will typically contain two kinds of information: (i) assignments of values to local and global variables; (ii) per-process execution buffer containing events that will eventually occur (for instance memory events or instructions waiting to be executed), where the order in which they will occur has not yet been determined.

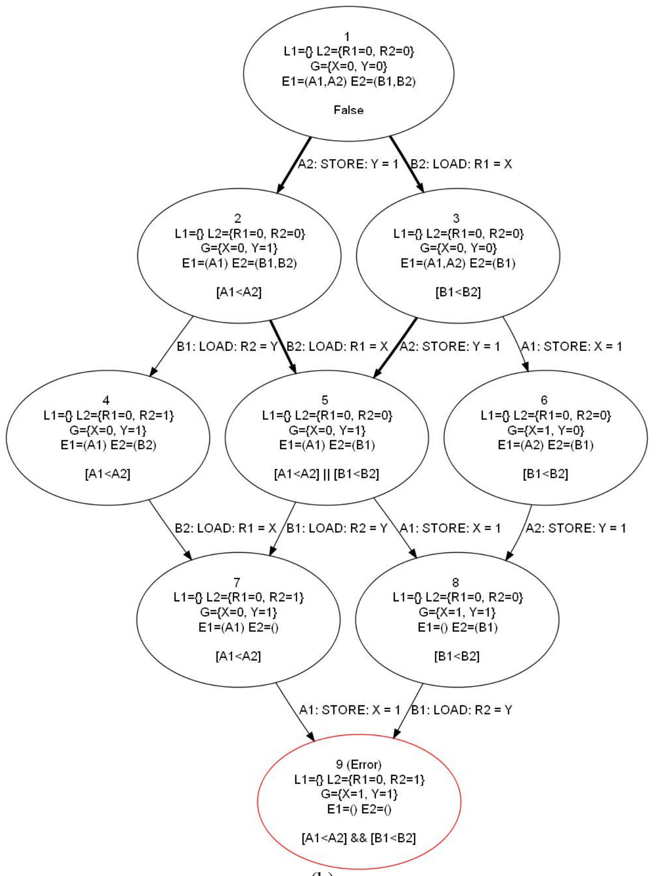
Computing Avoid Formulae Given a transition system and a specification, the goal of the inference algorithm is to infer fences that prevent execution of all traces leading to states that violate the specification (error states). One naive approach is to enumerate all (acyclic) traces leading to error states, and try to prevent each by adding appropriate fences. However, such enumeration does not scale to any practical program, as the number of traces can be exponential in the size of the transition system which is itself potentially exponential in the program length. Instead, our algorithm works on individual states and computes for each state an *avoid formula* that captures all the ways to prevent execution from reaching the state. Using the concept of an *avoidable transition* mentioned earlier, we can define the condition under which a state is avoidable. The avoid formula for a state σ considers all the ways to avoid all incoming transitions to σ by either: (i) avoiding the transition itself; or (ii) avoiding the source state of the transition. Since the transition system may contain cycles, the computation of avoid formulae for states in the transition system needs to be iterated to a fixed point.

Example Consider the simple program of Fig. 2(a). For this program, we would like to guarantee that $R1 \geq R2$ in its final state. For illustrative purposes, we consider a simple memory model where the stores to global memory are atomic and the only allowed relaxation is reordering data independent instructions. Fig. 2(b) shows part of the transition system built for the program running on this specific memory model. We only show states that can lead to an error state. In the figure, each state contains: (i) assignments to local variables of each process ($L1$ and $L2$), and the global variables G; (ii) the execution buffer of each process ($E1$ and $E2$); (iii) an avoid formula which we explain below.

The initial state (state 1) has $R1 = R2 = X = Y = 0$. There is a single error state where $R1 = 0$ and $R2 = 1$ (state 9). The avoid formula for each state is computed as mentioned earlier. For example, the avoid formula for state 2 is computed by taking the disjunction of avoiding the transition A_2 and avoiding the source state of the transition (state 1). To check whether A_2 is an avoidable transition from state 1, we check whether A_2 is executed out of order, and what are the alternative instructions that could have been executed by A instead. We examine the execution buffer $E1$ of state 1 and find all instructions that precede A_2. We find that A_2 is executed out of order, and that A_1 could have been

```
          R1 = R2 = X = Y = 0;
     A:                        B:
A1: STORE 1, X    ||    B1: LOAD Y, R1
A2: STORE 1, Y          B2: LOAD X, R2
```

(a)



(b)

Fig. 2. An example program (a) and its partial transition system (b). Avoidable transitions are drawn with thicker lines.

executed to avoid this transition. So, we generate the constraint $[A_1 < A_2]$ as a way to avoid the transition A_2. The meaning of the constraint is that this transition can be avoided if A_1 is executed before A_2. Since the source state (state 1) cannot be avoided, the avoid formula for state 2 is just $[A_1 < A_2]$. The constraint $[B_1 < B_2]$ for state 3 is obtained similarly.

For state 5, there are two incoming transitions: B_2 and A_2. Here, B_2 is taken out of order from state 2 and hence we generate the constraint $[B_1 < B_2]$. The constraint for the parent state 2 is $[A_1 < A_2]$, so the overall constraint becomes $[B_1 < B_2] \vee [A_1 < A_2]$. Similarly, we perform the computation for transition A_2 from state 3 which generates an identical constraint. The final avoid formula for state 5 is thus the conjunction of $[B_1 < B_2] \vee [A_1 < A_2]$ with itself. In other words, it is this exact formula. The transition from state 2 to state 4 is taken in order. Therefore, the transition itself cannot be avoided and the only way to avoid reaching 4 is through the avoid formula of its predecessor, state 2. For the error state 9, the two incoming transitions do not generate constraints as they are executed in-order. The overall constraint is thus generated as conjunction of the constraints of the predecessor states 7 and 8, and it is $[B_1 < B_2] \wedge [A_1 < A_2]$.

Because our example graph is acyclic, a single pass over the graph is sufficient. It is easy to check the formulas that appear in Fig. 2(b) indeed correspond to a fixed point. Since there is only one error state, the resulting overall constraint is the avoid constraint of that error state: $[A_1 < A_2] \wedge [B_1 < B_2]$.

Finally, this constraint can be implemented by introducing a store-store fence between A_1 and A_2 and a load-load fence between B_1 and B_2.

C. Memory Models

To demonstrate our fence inference algorithm on realistic relaxed memory models, we define and implement the model RLX that contains key features of modern memory models. According to the categorization of [2], summarized in Fig. 3, there are five such key features. The leftmost three columns in the table represent order relaxations. For instance, $W \rightarrow R$ means the model may reorder a write with a subsequent read from a different variable. The rightmost columns represent store atomicity relaxations - that is, whether a store can be seen by a process before it is globally performed. Our memory model supports four of these features, but precludes "reading other's writes early" and speculative execution of load instructions.

The memory model is defined operationally, in a design based on [13] and [14]. We represent instruction reordering by using an execution buffer, similar to the "reordering box" of [15] and the "local instr. buffer" of [14]. To support non-atomic stores we, like [13], split store operations into a "store; flush" sequence, and allow local load operations to read values that have not yet been flushed. This allows us to talk about the model purely in terms of reordering, without paying any additional attention to the question of store atomicity.

Barring speculative execution of loads, RLX corresponds to Sun SPARC v9 RMO and is weaker than the SPARC v9 TSO and PSO models. RLX is strictly weaker than the IBM 370. Since RLX is weaker than these models, any fences that we infer for correctness under RLX are going to guarantee correctness under these models.

Our framework allows to instantiate models stronger than RLX, by disabling some of the relaxations in RLX. In fact, the framework supports any memory model that can be expressed using a bypass table (similar to [14] and the "instruction reordering table" of [13]). This enables us to experiment with fence inference while varying the relaxations in the underlying memory model. In Section V, we show how different models lead to different fence placements in practical concurrent algorithms, demonstrating the importance of automatic inference.

IV. Inference Algorithm

In this section, we describe our fence inference algorithm. Due to space restrictions, the description is mostly informal. The full technical details can be found in [16].

Relaxation	W → R Order	W → W Order	R → RW Order	R Others' W Early	R Own W Early
SC					✓
IBM 370	✓				
TSO	✓				✓
PSO	✓	✓			✓
Alpha	✓	✓	✓		✓
RMO	✓	✓	✓		✓
PowerPC	✓	✓	✓	✓	✓

Fig. 3. Categorization of relaxed memory models, from [2].

A. Preliminaries

We define a program P in the standard way, as a tuple containing an initial state $Init$, the program code $Prog_i$ for each processor, and an initial statement $Start_i$. The program code is expressed in a simple assembly-like programming language, which includes load/store memory operations, arbitrary branches and compare-and-swap operations. We assume that all statements are uniquely labeled, and thus a label uniquely identifies a statement in the program code, and denote the set of all program labels by *Labs*.

Transition Systems A transition system for a program P is a tuple $\langle \Sigma_P, T_P \rangle$, where Σ_P is a set of states, T_P is a set of labeled transitions $\sigma \xrightarrow{l} \sigma'$. A transition is in T_P if $\sigma, \sigma' \in \Sigma_P$ and $l \in$ *Labs*, such that executing the statement at l results in state σ'. The map $enabled\colon \Sigma_P \to \mathcal{P}(Labs)$ is tied to the memory model and specifies which transitions may take place under that model.

Dynamic Program Order Much of the literature on memory models (e.g. [11], [12], [17]) bases the model's semantics on the concept of *program order*, which is known a priori. This is indeed the case for loop-free or statically unrolled programs. For programs that contain loops, Shen et. al show in [13] that such an order is not well defined, unless a memory model is also provided. Furthermore, for some memory models the program order may depend on the specific execution.

To accommodate programs with loops, we define a *dynamic program order*. This order captures the program order at any point in the execution. For a given state σ and a process p, we write $l_1 <_{\sigma,p} l_2$ when l_1 precedes l_2 in the dynamic program order. The intended meaning is that in-order execution from state σ would execute the statement at l_1 before executing the statement at l_2.

B. An Algorithm for Inferring Ordering Constraints

Given a finite-state program P and a safety specification S, the goal of the algorithm is to infer a set of ordering constraints that prevent all program executions violating S and can be implemented by fences.

Avoidable Transitions and Ordering Constraints The ordering constraints we compute are based on the concept of an *avoidable transition* — a transition taken by the program that could have been *prohibited* by some fence. This captures the intuition of a transition that was taken out of order. To identify such transitions we use the dynamic program order: a transition $t = \sigma \xrightarrow{l_t} \sigma'$ is avoidable if there exists some l_1 such that $l_1 <_{\sigma,p} l_t$.

With every pair of labels $l_1, l_2 \in$ *Labs* we associate a proposition $[l_1 \prec l_2]$. We call such a proposition an *ordering constraint*. We define a *constraint formula* as a propositional formula over ordering constraints. For each transition $t = \sigma \xrightarrow{l_t} \sigma'$ we then define the formula $prevent(t) = \bigvee\{[l_1 \prec l_t] \mid l_1 <_{\sigma,p} l_t\}$. Intuitively, $prevent(t)$ is the formula that captures all possible ordering constraints that would prohibit the execution of t by the program. Note that if t is not avoidable, this is an empty disjunction and $prevent(t) = false$.

Algorithm 1: Fence Inference

```
   Input: Program P, Specification S
   Output: Program P' satisfying S
1  compute $\langle \Sigma_P, T_P \rangle$
2  $avoid(Init) \leftarrow false$
3  foreach state $\sigma \in \Sigma_P \setminus \{Init\}$ do
4      $avoid(\sigma) \leftarrow true$
5  workset$\leftarrow \Sigma_P \setminus \{Init\}$
6  while workset is not empty do
7      $\sigma \leftarrow$ select and remove state from workset
8      $\varphi \leftarrow avoid(\sigma)$
9      foreach transition $t = (\mu \longrightarrow \sigma) \in T_P$ do
10         $\varphi \leftarrow \varphi \wedge (avoid(\mu) \vee prevent(t))$
11     if $avoid(\sigma) \not\equiv \varphi$ then
12         $avoid(\sigma) \leftarrow \varphi$
13         add all successors of $\sigma$ in $\Sigma_P$ to workset
14 $\psi \leftarrow \bigwedge\{avoid(\sigma) \mid \sigma \not\models S\}$
15 return implement(P, $\psi$)
```

Inference The algorithm operates directly on program states. For every state σ in the program's transition system, the algorithm computes a constraint formula $avoid(\sigma)$ such that satisfying it prevents execution from reaching σ. The computed formula $avoid(\sigma)$ captures all possible ways to prevent execution from reaching σ by forbidding avoidable transitions.

The algorithm computes a fixed point of avoid constraints for all states in the program's transition system. First, we build the transition system $\langle \Sigma_P, T_P \rangle$ of the program. For $\sigma = Init$, we initialize $avoid(\sigma)$ to *false*. For all other states, we initialize it to *true*. We then add all states to the workset. The algorithm proceeds by picking a state from the workset, and computing the new avoid constraint for the state. A state can only be avoided by avoiding all incoming transitions (a conjunction). To avoid the transition, we must (i) consider all possible ways to avoid the transition from the predecessor state (by using $prevent(t)$); *or* (ii) avoid the predecessor state, by using its own avoid constraint. (see line 10 of the algorithm).

As shown in line 11 every such computation step requires comparing two boolean formulas for equality. While in general NP-hard, this is not a problem in practice due to the structure of our formulas and their relatively modest size.

When a fixed point is reached, the algorithm computes the overall constraint ψ by taking the conjunction of avoid constraints for all error states. Any implementation satisfying ψ is guaranteed to avoid all error states, and thus satisfy the specification. Finally, the algorithm calls the procedure `implement(P,`ψ`)` which returns a program that satisfies ψ.

Ensuring Termination In cases where the transition system is an acyclic graph (e.g. transition systems for spinloop-free programs), we can avoid performing the fixed point computation altogether. If the states are topologically sorted, the computation can be completed with a single linear pass over the transition system. In the general case, we can show the set of mappings between states and constraints forms a finite lattice and our function is monotonic and continuous. Thus convergence is assured.

Safety and Maximal Permissiveness Given a program P and a specification S, the avoid formula φ computed by Algorithm 1 is the *maximally permissive* avoid formula such that all traces of P satisfying φ are guaranteed to satisfy S. More formally, we say a constraint formula admits a transition $t = \sigma \xrightarrow{l_t} \sigma'$ if there exists an assignment $\alpha \vDash \varphi$ so that every proposition of the form $v = [l_1 \prec l_t]$ where $l_1 <_{\sigma,p} l_t$ we have $[\![v]\!]_\alpha = false$. Here $[\![v]\!]_\alpha$ is the value of proposition v in the assignment α. We can lift this definition of *admits* from transitions to program traces. Then if $\varphi \neq false$ it only admits traces that satisfy S, but for any $\psi \neq \varphi$ such that $\varphi \Rightarrow \psi$, there exists a trace π of P that reaches σ such that ψ admits π, but $\sigma \nvDash S$.

C. Fence Inference

Our algorithm computes a maximally permissive constraint formula ψ. We can then use a standard SAT-solver to get assignments for ψ, where each assignment represents a set of constraints that enforces correctness. Since for a set of constraints C, a superset C' cannot be more efficiently implemented, we need only consider minimal (in the containment sense) sets.

An orthogonal problem is to define criteria that would allow us to select optimal fences that enforce one of those sets. In our work, we focus on a simple natural definition using set containment: a fence placement is a set of program labels where fences are placed and we say that a placement P_1 is better than P_2 when $P_1 \subseteq P_2$.

Given a minimal assignment C for the formula ψ, for each satisfied proposition $[l_1 \prec l_2]$, we can insert a fence either right after l_1 or right before l_2, thus getting a correct placement of fences. We can try this for all minimal assignments of ψ, and select only the minimal fence placements. This procedure can be improved by defining a formula ξ s.t. every proposition in ψ is replaced with $after(l_1) \vee before(l_2)$. Here, $after(l)$ and $before(l)$ map labels to a new set of propositions, so that if l_2 appears immediately after l_1 in the program, then $after(l_1) = before(l_2)$. Then, our fence placements will be the minimal assignments to ξ. This allows us to directly apply a SAT-solver and consider fewer fence placements.

Of course this local approach will not guarantee a minimal placement of fences because there can be many ways to implement a constraint $[l_1 \prec l_2]$ aside from inserting a fence immediately after l_1 or before l_2. For instance, if $l_1, ...l_4$ appear in this order in the program, and $\psi = [l_1 \prec l_4] \wedge [l_2 \prec l_3]$ then we can implement ψ by a single fence between l_2 and l_3. More precise and elaborate implementation strategies are possible if the program's control flow graph is taken into account. However, in our experiments we found the simple local fence placement strategy to produce optimal results.

V. Experiments

We have implemented our algorithm in a tool called FENDER. Our tool takes as input a description of a memory model, a program and a safety specification. The tool then automatically infers the necessary memory fences.

A. Methodology

We experiment with FENDER by varying the following:

(i) Input Algorithm - we experiment with five concurrent data structures and one mutual exclusion algorithm.
(ii) Client Program - we experiment with clients of varying size and complexity.
(iii) Memory Model - we experiment with 3 relaxed models and the sequentially consistent model as a baseline.
(iv) Specification - in some benchmarks, there is more than one reasonable specification.
(v) Bound on the execution buffer, when required.

Algorithms We applied our tool to various challenging state-of-the-art concurrent algorithms:

- *MSN*: Michael&Scott's lock-free queue [18].
- *LIFO WSQ*: LIFO idempotent work-stealing queue [19].
- *Chase-Lev WSQ*: Chase&Lev's work-stealing queue [9].
- *Dekker*: Dekker's mutual exclusion [20].
- *Treiber*: Treiber's lock-free stack [21].
- *VYSet*: Vechev&Yahav's concurrent list-based set [22].

Clients For each algorithm, we ran FENDER with several clients. Our tool permits exhaustive exploration of bounded clients that consist of a (bounded) sequence of initialization operations followed by (bounded) sequences of operations performed in parallel. A client typically consists of 2 or 3 processes, where each process invokes several data structure operations. Below, we use the term "program" to refer to the combination of an algorithm and a client.

Memory Models As noted earlier, our RLX model is equivalent to SPARC RMO without support for speculation. Our framework can instantiate stronger models, and in our experiments, we infer fences under four memory models: RMO, PSO, TSO, and as a reference, SC, the sequentially consistent model. The models RMO, PSO and TSO implement three different sets of relaxations as described in [2]. All three implement the "read own writes early" relaxation. RMO implements the $W \rightarrow R$, $W \rightarrow W$ and $R \rightarrow RW$ relaxations. PSO removes the $R \rightarrow RW$ relaxation and TSO additionally removes the $W \rightarrow W$ relaxation.

Specification We consider safety specifications realized as state invariants on the program's final state. To write an invariant, for most algorithms, we observed the results a specification of sequential consistency would produce and then write invariants that are implied by this specification. In

	Initial State	Client	$\|E\|$ Bnd	Time (sec.)	States	Edges	#C
MSN	empty	e|d	∞	0.83	1219	2671	2
	empty	e|e	∞	1.78	4934	12670	1
	empty	ee|dd	∞	5.21	24194	61514	3
	empty	ed|ed	∞	13.05	86574	242822	2
	empty	ed|de	∞	9.26	59119	167067	4
	empty	e|e|d	∞	31.43	233414	653094	3
ChaseLev WSQ	empty	pppt(tpt|sss)	∞	97.22	386283	1030857	-
	empty	tttt(ptt|sss)	∞	255.5	1048498	2819355	-
	empty	pppt(ttp|sss)	∞	90.28	281314	878880	-
	empty	tttt(tpp|sss)	∞	355.95	1325858	4150650	-
	empty	tttp(tptp|ss)	∞	37.98	280396	698398	-
”LIFO” WSQ	2/2	tp|ss	∞	0.69	2151	3190	2
	2/2	tpt|ss	∞	1.94	9721	16668	2
	2/2	ptp|ss	∞	11.41	89884	195246	3
	2/2	ptt|ss	∞	11.31	85104	198353	4
	1/1	ptt|ss	∞	4.07	23913	48997	4
Dekker	-	-	1	0.64	1388	2702	2
	-	-	10	2.13	7504	14477	2
	-	-	20	2.71	13879	26422	2
	-	-	50	5.99	33004	62257	2
Treiber	empty	p|t	∞	1	71	93	2
	empty	pt|tp	∞	1.02	3054	6190	2
	empty	pp|tt	∞	0.6	1276	2250	2
VYSet	empty	ar|ra	10	1.98	4079	6247	2
	empty	aa|rr	10	4.56	20034	31623	2
	empty	ar|ar	10	2.19	6093	9905	2
	empty	aaa|rrr	10	7.98	41520	66533	2

TABLE II
EXPERIMENTAL RESULTS FOR THE RMO MODEL

this context, sequential consistency refers not to the memory model, but to the high level specification that an algorithm should satisfy. In some experiments we also used additional, weaker specifications.

Bound on the Execution Buffer As recently shown in [23], the reachability problem for weak memory models is, depending on the model, either undecidable or non-primitive recursive even for finite-state programs. To avoid this problem we add a stronger condition and require the execution buffers to be bounded. In four of our benchmarks this was the natural behavior, and in the other two we've had to enforce a bound.

Experimental Setup Experiments were performed on an IBM xSeries 336 with 4 Intel Xeon 3.8Ghz processors, 5GB memory, running a 64-bit Red Hat Enterprise Linux. Tab. II contains performance metrics for RMO, the most relaxed memory model that we considered.

B. Results

A summary of our experimental results is shown in Tab. II. For each data structure, several parallel clients were used. For each client, the “Initial” and ”Client” columns represent the initial state of the data structure and the operations performed by the client respectively. “e” represents an *enqueue* operation, “d” a *dequeue*, “p” *put*, “s” *steal*, “a” *add* and “r” *remove*. The “|E|” column represents the bound on the length of execution buffers, and “#C” the number of constraints in a minimal solution to the avoid formula for that client. Since for *Chase-Lev* the constraint formula was solved only for the conjunction of all clients, individual “#C” values are not given. The “Time” column shows the total analysis time. This includes the state exploration time, the constraint inference time and the SAT-solving time. Note that in all cases the solving component was negligible.

In Tab. III we show a comparison of the performance of FENDER for different memory models it supports. On average the number of states for PSO was ≈ 4.5 times smaller and for TSO ≈ 40 times smaller than for RMO.

Chase-Lev Work Stealing Queue For this data structure, we ran an exhaustive set of clients with two bounds: (i) all clients were of the form of 4 initializer operations, followed by a parallel section with $5 > X > 3$ invocations by the owner, and $6 - X$ steal invocations by another process. (ii) If a particular client's state space exceeded 2.5 million states, it was terminated and discarded. In Tab. II we show representative clients that produced useful constraints. In those experiments, FENDER inferred a set of 9 constraints which can be implemented using the 6 fences of Fig. 1. In particular, the fence between lines 9 and 10 in `expand()` also prevents the reordering of the store on line 10 with the stores on lines 8 an 6. Under PSO, we are left with 6 constraints and 3 fences—all of the fences in `steal()` are no longer needed. Even under TSO, one fence still remains necessary—it is the store-load fence between lines 4 and 5 in the `take()` operation.

Michael-Scott Queue For MSN FENDER inferred all 3 required fences under RMO. The placement for this algorithm in [7] contained 7 fences, however, 2 of these are the result of [7] allowing extra speculation, and 2 are not required in our model due to conservative memory allocation. Under PSO a single fence was inferred, and under TSO no fences are required.

Idempotent Work-Stealing The reference placement in [19] is phrased only in terms of constraints, and requires 5 constraints. Under RMO, FENDER produced 4 constraints which are a subset of those 5. The one constraint not inferred is, again, only required because of possible speculation.

Dekker's Algorithm It is well known that Dekker's algorithm requires a fence in the entry section and a fence at the end of the section (to preserve semantics of critical section). In our experiments, FENDER successfully inferred the required fences. Under RMO and PSO both fences were inferred, and under TSO, the tool inferred only the entry section fence. This is consistent with the reference placement appearing in Appendix J of [11].

C. Discussion

In our experiments, we observe that the fences inferred by FENDER are quite tricky to get manually. For some of the algorithms, there are known correct fence assignments, and for these we show that FENDER derives all necessary fences for our memory models with a small number of clients driving the algorithm. For most of our benchmarks, a bound on the execution buffer was not required. In the two cases where it was required, all fences were obtained with a small bound.

A recurring theme in our results was that several different maximally permissive constraint sets could be

	Initial	Client	$\|E\|$	RMO			PSO			TSO			SC	
			Bound	States	Edges	#C	States	Edges	#C	States	Edges	#C	States	Edges
MSN	empty	e|d	∞	1219	2671	2	455	743	1	228	316	0	146	180
	empty	e|e	∞	4934	12670	1	2678	6354	1	586	994	0	252	328
	empty	ee|dd	∞	24194	61514	3	7025	13689	2	1724	2512	0	1029	1325
	empty	ed|ed	∞	86574	242822	2	15450	35362	2	2476	3972	0	1538	2126
	empty	ed|de	∞	59119	167067	4	11023	24362	2	2570	4010	0	1541	2073
	empty	e|e|d	∞	233414	653094	3	51990	119050	2	9638	16822	0	4928	7632
Chase-Lev	empty	pppt(tpt|sss)	∞	386283	1030857	-	74533	256613	-	12348	20004	-	4961	6740
WSQ	empty	tttt(ptt|sss)	∞	1048498	2819355	-	124455	255390	-	6418	9380	-	3101	4069
	empty	pppt(ttp|sss)	∞	281314	878880	-	66960	241814	-	10564	16317	-	4199	5700
	empty	tttt(tpp|sss)	∞	1325858	4150650	-	361855	1080835	-	9878	13956	-	3473	4537
	empty	tttp(tptp|ss)	∞	280396	698398	-	29573	54696	-	9197	14499	-	4760	6455
"LIFO"	2/2	tp|ss	∞	2151	3190	2	882	1171	1	676	852	0	570	694
WSQ	2/2	tpt|ss	∞	9721	16668	2	3908	5811	1	2256	3116	0	1410	1786
	2/2	ptp|ss	∞	89884	195246	3	31289	64133	3	4045	5688	0	2317	3007
	2/2	ptt|ss	∞	85104	198353	4	29920	62020	3	4130	5987	0	2198	2866
	1/1	ptt|ss	∞	23913	48997	4	9976	18002	3	2353	3269	0	1314	1654
Dekker	-	-	1	1388	2702	2	1388	2702	2	489	674	1	388	490
	-	-	10	7504	14477	2	7504	14477	2	2560	3750	1	388	490
	-	-	20	13879	26422	2	13879	26422	2	4845	7115	1	388	490
	-	-	50	33004	62257	2	33004	62257	2	11770	17210	1	388	490
Treiber	empty	p|t	∞	71	93	2	71	93	2	43	48	0	36	38
	empty	pt|tp	∞	3054	6190	2	3041	6167	2	407	609	0	392	482
	empty	pp|tt	∞	1276	2250	2	1276	2250	2	325	407	0	270	323
VYSet	empty	ar|ra	10	4079	6247	2	4079	6247	2	1088	1308	0	1088	1308
	empty	aa|rr	10	20034	31623	2	20034	31623	2	1168	1411	0	1168	1411
	empty	ar|ar	10	6093	9905	2	6093	9905	2	1671	1968	0	1671	1968
	empty	aaa|rrr	10	41520	66533	2	41520	66533	2	3311	4072	0	3311	4072

TABLE III
EXPERIMENTAL RESULTS FOR DIFFERENT MEMORY MODELS

derived from the constraint formula. However, in all cases, all of those sets represented one "natural" solution. The reason for the appearance of those apparently different solutions involves data dependencies. Consider the simple example program shown on the right. Assume that the constraint $[l_1 \prec l_3]$ must be enforced in any execution. However, if $[l_1 \prec l_2]$ is enforced, then it is impossible to reorder l_3 with l_1. Due to a data dependency, l_2 must come before l_3, and we get the order $\sigma_1 \xrightarrow{l_2} \sigma_2 \xrightarrow{l_3} \sigma_3 \xrightarrow{l_1} \sigma_4$ in which the first transition violates $[l_1 \prec l_2]$. Thus, our constraint formula will necessarily contain the disjunction $[l_1 \prec l_2] \vee [l_1 \prec l_3]$. It is an interesting question whether there exists an input algorithm which permits several *substantially* different constraint sets.

```
1   STORE Z = 1
2   LOAD R = X
3   STORE Y = R
```

As expected, when we ran the tool with more restricted memory models, the number of required fences decreases. For example, the move from PSO to TSO disables reordering of independent stores and hence all constraints between stores to different locations are not required.

VI. Related Work

Earlier we discussed work directly related to fence inference, that is [7], [8]. Additional related work includes:

Explicit-State Model Checking The works closest to ours in the way they explore the state space for a weak memory model are [15] and [24]. Both describe explicit-state model checking under the Sparc RMO model, but neither uses it for inference.

Delay Set Analysis A large body of work relies on the concepts of *delay set* and *conflict graph* of [25] for reasoning about relaxed memory models. In particular, the Pensieve project [26], [27], [28] implements fence synthesis based on delay set analysis. This kind of analysis is, however, necessarily more conservative than ours since it prevents any *potential* specification violations due to non-SC execution, and is not appropriate for highly concurrent algorithms.

Verification Approaches In [29] and [30] algorithms are presented that can find violations of sequential consistency under the TSO and PSO memory models. Those algorithms find violations based purely on sequentially consistent executions, thus making them very efficient. However, just like delay set analysis, this is often needlessly conservative. Another approach to verification is to try to establish a property which ensures the program remains correct under relaxed models. The most common such property is *data-race freedom*, as for data-race free programs the "fundamental property of memory models" [31] ensures that there can be no sequentially inconsistent executions. In our work we deal with programs that do not satisfy such properties. Further, none of those works supports fence inference for programs that are found to violate SC.

Inference of Synchronization In [32], [22], a semi-automated approach is used to explore a space of concurrent garbage collectors and linearizable data-structures. These works do not support weak memory models. In [33] a framework similar to ours is used to infer minimal synchronization. However the technique used there enumerates traces explicitly, which does not scale in our setting and thus cannot be applied as-is.

Effect Mitigation Several works have been published on mitigating the effect of memory fences [34], [35] and making synchronization decisions during runtime [36]. Those architectural improvements are complementary to our approach.

VII. Summary and Future Work

We presented a novel fence inference algorithm and demonstrated its practical effectiveness by evaluating it on various challenging state-of-the-art concurrent algorithms. In future work, we intend to improve the tool's scalability and add support for more memory models. Another direction we intend to pursue is memory model abstraction and fence inference under abstraction. This will allow us to avoid bounding the execution buffer and make our algorithm more suitable for more general input programs.

VIII. Acknowledgements

We would like to thank Maged Michael for his valuable comments and advice during the preparation of this paper.

References

[1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kauffman, Feb. 2008.

[2] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, pp. 66–76, 1995.

[3] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess program," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, 1979.

[4] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[5] M. M. Michael and M. L. Scott, "Correction of a memory management method for lock-free data structures," Tech. Rep., 1995.

[6] S. Burckhardt, R. Alur, and M. M. K. Martin, "Bounded model checking of concurrent data types on relaxed memory models: A case study," in *CAV*, 2006, pp. 489–502.

[7] ——, "Checkfence: checking consistency of concurrent data types on relaxed memory models," in *PLDI*, 2007, pp. 12–21.

[8] T. Q. Huynh and A. Roychoudhury, "Memory model sensitive bytecode verification," *Form. Methods Syst. Des.*, vol. 31, no. 3, 2007.

[9] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *SPAA*, 2005, pp. 21–28.

[10] M. M. Michael, "Safe memory reclamation for dynamic lock-free objects using atomic reads and writes," in *PODC*, 2002, pp. 21–30.

[11] I. SPARC International, *The SPARC architecture manual (version 9)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[12] A. Adir, H. Attiya, and G. Shurek, "Information-flow models for shared memory with an application to the powerpc architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 5, pp. 502–515, 2003.

[13] X. Shen, Arvind, and L. Rudolph, "Commit-reconcile & fences (crf): a new memory model for architects and compiler writers," *SIGARCH Comput. Archit. News*, vol. 27, no. 2, pp. 150–161, 1999.

[14] Y. Yang, G. Gopalakrishnan, and G. Lindstrom, "Umm: an operational memory model specification framework with integrated model checking capability," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 5-6, pp. 465–487, 2005.

[15] S. Park and D. L. Dill, "An executable specification and verifier for relaxed memory order," *IEEE Transactions on Computers*, vol. 48, 1999.

[16] M. Kuperstein, M. Vechev, and E. Yahav, "Automatic inference of memory fences: Technical report," Technion, TR, 2010. [Online]. Available: http://www.cs.technion.ac.il/~mkuper/autoinf.pdf

[17] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-cc multiprocessor machine code," in *POPL*, 2009, pp. 379–391.

[18] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *PODC*, 1996, pp. 267–275.

[19] M. M. Michael, M. T. Vechev, and V. A. Saraswat, "Idempotent work stealing," in *PPoPP*, 2009, pp. 45–54.

[20] E. Dijkstra, "Cooperating sequential processes, TR EWD-123," Technological University, Eindhoven, Tech. Rep., 1965.

[21] R. Treiber, "Systems programming: Coping with parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, Apr. 1986.

[22] M. Vechev and E. Yahav, "Deriving linearizable fine-grained concurrent objects," in *PLDI*, 2008, pp. 125–135.

[23] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, "On the verification problem for weak memory models," in *POPL*, 2010, pp. 7–18.

[24] B. Jonsson, "State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version)," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 65–71, 2008.

[25] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, 1988.

[26] J. Lee and D. A. Padua, "Hiding relaxed memory consistency with a compiler," *IEEE Trans. Comput.*, vol. 50, no. 8, pp. 824–833, 2001.

[27] X. Fang, J. Lee, and S. P. Midkiff, "Automatic fence insertion for shared memory multiprocessing," in *ICS*, 2003, pp. 285–294.

[28] Z. Sura, C. Wong, X. Fang, J. Lee, S. Midkiff, and D. Padua, "Automatic implementation of programming language consistency models," *LNCS*, vol. 2481, p. 172, 2005.

[29] S. Burckhardt and M. Musuvathi, "Effective program verification for relaxed memory models," in *CAV*, 2008, pp. 107–120.

[30] J. Burnim, K. Sen, and C. Stergiou, "Sound and complete monitoring of sequential consistency in relaxed memory models," Tech. Rep. UCB/EECS-2010-31. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-31.html

[31] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun, "A theory of memory models," in *PPoPP*. ACM, 2007, pp. 161–172.

[32] M. T. Vechev, E. Yahav, D. F. Bacon, and N. Rinetzky, "Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors," in *PLDI*, 2007, pp. 456–467.

[33] M. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," in *POPL '10*, 2010.

[34] O. Trachsel, C. von Praun, and T. Gross, "On the effectiveness of speculative and selective memory fences," *IPDPS*, p. 15, 2006.

[35] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: performance-transparent memory ordering in conventional multiprocessors," in *ISCA*, 2009, pp. 233–244.

[36] C. von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu, "Conditional memory ordering," in *ISCA*, 2006, pp. 41–52.