Automatic Inference of Memory Fences

Michael Kuperstein, Martin Vechev, Eran Yahav

Abstract

We addresses the problem of automatic verification and fence inference in concurrent programs running under relaxed memory models. Modern architectures implement relaxed memory models in which memory operations may be reordered and executed non-atomically. Instructions called *memory fences* are provided to the programmer, allowing control of this behavior. To ensure correctness of many algorithms, the 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. It is our goal to help automate the fence insertion process.

We present an algorithm for automatic inference of memory fences in concurrent programs, relieving the programmer from this complex task. Given a finite-state program, a safety specification and a description of the memory model our algorithm 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. These constraints are then realized as additional fences in the input program.

We implemented our approach in a pair of tools called FENDER and BLENDER and used them to infer correct and efficient placements of fences for several non-trivial algorithms, including practical mutual exclusion primitives and concurrent data structures.

1 Introduction

In 1979, in his seminal paper "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs" [20], Leslie Lamport defined the "sequential consistency" (SC) criterion for correctness of multiprocessor computers. Such a computer is called sequentially consistent if:

The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual processor appear in the sequence in the order specified by its program.

It was Lamport's intent that any correct multiprocessor computer implementation must meet this criterion. However, modern hardware architectures are not, in fact, sequentially consistent. Instead, they implement so-called "relaxed" (or "weak") memory models (RMMs) [1]. These models enable improved hardware performance compared to sequentially consistent hardware [15]. This is achieved by allowing the CPU and memory subsystems to perform memory operations out of order and non-atomically. Unfortunately, this behavior poses an additional burden on the programmer. Even when the underlying architecture is sequentially consistent, highly-concurrent algorithms are notoriously hard to get right [26]. When programming for an architecture that implements an RMM, programmers must also reason about executions that have no sequential analogue. This reasoning is non-intuitive and may lead to subtle concurrency bugs.

To allow programmers avoid non-sequentially consistent executions, architectures provide special *memory fence* (also known as *memory barrier*) instructions. Very informally, a fence instruction restricts the CPU and memory subsystem's ability to reorder operations, thus eliminating some undesired non-SC executions. Finding a *correct and efficient* placement of memory fences for a given concurrent program is a challenging task. Using too many fences (over-fencing) hinders performance, while using too few fences (under-fencing) may allow unexpected incorrect executions to occur. Manually balancing between over- and under-fencing is very difficult, time-consuming and error-prone (cf. [16, 5, 6]). Furthermore, the process of finding fences has to be repeated whenever the algorithm changes, and whenever it is ported to a different architecture.

As an example, consider the problem of implementing the Chase-Lev work-stealing queue [11] ("CL") 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 [3] and X10 [10]. Fig. 1 shows an implementation of this data structure in C-like pseudo-code. For now, ignore the fence instructions that appear on unnumbered lines. CL maintains an expandable array of items called *wsq* and two indices *top* and *bottom*, initialized to 0. The queue is considered empty when $top \geq bottom$ When the queue is not empty, $top\%(queue \rightarrow size)$ points to the oldest element in the queue, while $bottom\%(queue \rightarrow size)$ points one past the newest element. 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 may call `steal()` to take items out from the opposite end. The queue can be dynamically expanded in response to a `push()` when additional space is required to store the item. This is done by the `push()` operation invoking the `expand()` procedure. 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, cf. [25]).

We would like to guarantee that there are no out of bounds array accesses, no items are lost (by being overwritten before being read), and no "phantom" items are read after being removed. All these properties hold for the CL queue under the sequentially consistent memory model. However, they may be violated when it is used under a relaxed model.

Under weak memory models, e.g. the SPARC RMO [30] memory model, some of the memory operations in the code may be executed out of order. Tab. 1 shows possible RMO re-orderings 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. The last column shows the type of fence that can be used

```
typedef struct {
  long size;
  int  *ap;
} item_t;

long top, bottom;
item_t *wsq;
```

```
void push(int task) {
 long b = bottom;
 long t = top;
 item_t* q = wsq;
 if (b-t ≥ q→size-1){
   q = expand();
 }
 q→ap[b % q→size]=task;
 fence("store-store");
 bottom = b + 1;
}
```

```
int take() {
 long b = bottom - 1;
 item_t* q = wsq;
 bottom = b;
 fence("store-load");
 long t = top;
 if (b < t) {
   bottom = t;
   return EMPTY;
 }
 task = q→ap[b % q→size];
 if (b > t)
   return task;
 if (!CAS(&top, t, t+1))
   return EMPTY;
 bottom = t + 1;
 return task;
}
```

```
int steal() {
 long t = top;
 fence("load-load");
 long b = bottom;
 fence("load-load");
 item_t* q = wsq;
 if (t ≥ b)
   return EMPTY;
 task=q→ap[t % q→size];
 fence("load-store");
 if (!CAS(&top, t, t+1))
   return ABORT;
 return task;
}
```

```
item_t* expand() {
  int  newsize = wsq→size * 2;
  int* newitems = (int *) malloc(newsize*sizeof(int));
  item_t *newq =  (item_t *)malloc(sizeof(item_t));
  for (long i = top; i < bottom; i++) {
    newitems[i % newsize] = wsq→ap[i % wsq→size];
  }
  newq→size = newsize;
  newq→ap = newitems;
  fence("store-store");
  wsq = newq;
  return newq;
}
```

Figure 1: Pseudo-code of the Chase-Lev work stealing queue

to prevent the undesirable reordering. Informally, the type describes what kinds of memory 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.

For a more detailed example of the effect the memory model has on execution, we consider the failure described in line 2 of Tab. 1. This corresponds to a reordering of operations at lines 4 and 5 in the `take()` method: if these two lines are reordered, the read from *top* is executed before the write to *bottom*. The failure scenario involves one process running the `steal()` method in parallel to another processes running a sequence of `take();push();take();push()` as follows:

1. Initially the queue has one item with $top = 0$ and $bottom = 1$.
2. A `take()` reads *top* and gets preempted before executing line 6.
3. An entire `steal()` executes, correctly returns the item at index 0, and advances *top* to 1.
4. The `take()` resumes and succeeds, returning the same item as the previous `steal()`, setting *bottom* to 0.
5. A complete `push()` now pushes some item i.
6. A complete `take()` executes and returns `EMPTY` instead of item i.
7. A complete `push()` executes and overwrites item i (losing item i).

To guarantee correctness under RMO, the programmer can try to manually insert fences that avoid undesirable behavior. As an alternative to placing fences based purely on her intuition, the programmer may also use a tool such as CheckFence [6] that can check the correctness of a given fence placement. However, 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 example. In practice, localizing a failure to a single reordering is challenging and time consuming as a single failure trace might include multiple instances of non-SC behavior. Furthermore, a single reordering can be exhibited as multiple failures, and it is sometimes hard to identify the cause underlying an observed failure trace.

In a nutshell, the programmer is required to manually produce Tab. 1: 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

#	**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 1: Potential reorderings of operations in the Chase-Lev algorithm of Fig. 1 running on the RMO memory model

model. Even a subtle change in the algorithm may require a complete re-examination.

It is easy to see that the process of manual fence inference does not scale. In this paper we present an algorithm that *automatically* infers *correct and efficient* fence placements for *finite-state* programs. 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 implemented a relaxed memory model that supports key features of several modern RMMs.

Requiring the input program to be finite-state means we must overcome several challenges for the algorithm to be practical. First, this requirement means the algorithm, taken as is, is not suitable for fence inference in open systems (such as library implementations). This is in contrast to our goal to apply the algorithm to concurrent data structures. To formally verify that a data structure meets a specification (and, consequentially, to infer a correct fence placement), one generally needs to verify the "most general client" which is usually not finite-state. We deal with this in a manner similar to other related work in the field (e.g., Burckhardt et al. [6]) by using representative clients. Another problem is that even if a program is finite state under sequential consistency it will often not be finite-state under a relaxed model. As this phenomenon is common in practice, a direct implementation of our algorithm fails to infer fences for many interesting programs. To solve this problem we developed the concept of *abstract memory models* (AMMs). Very informally, an abstract memory model is an over-approximation of a relaxed memory model, in the sense that any program behavior possible in the RMM is also possible in the abstract model. Our abstract memory models are designed so as a program that is finite-space under SC remains finite state under the AMM. By utilizing AMMs, we can use our algorithm for any program that is finite-space under SC. More detail on AMMs can be found in [19] and [17].

In this paper, we describe an algorithm that automatically infers a correct and efficient placement of memory fences in finite-state concurrent programs. The paper is based on work previously published as [18] and [17].

2 Fence Inference

2.1 Constraint Generation

We first present our inference algorithm in a general setting, without instantiating it for a specific memory model. We then prove that when properly instantiated, it is correct and optimal.

Goal The input to the algorithm is a finite-state program P, a safety specification S, and an operational description of the memory model M. We assume that P satisfies S under sequential consistency but not necessarily under M. The output of the algorithm is a new program P', that satisfies S under M, which is obtained by adding memory fences to P. For "reasonable" memory models, this problem always has a trivial solution, as placing a fence between every two memory operations will reduce the possible executions to those allowed under SC. Therefore, we also add an optimality constraint: we would like the program P' to have performance as close to the original program as possible. In other words, since there may be many possible ways to "fix" P by adding fences, we want to choose only the best solution(s).

Algorithm Structure Our algorithm follows these three steps:

1. Construct the transition system for P under M.
2. Find the set of "error states" E violating S in the transition system.
3. Compute a set of program locations in P s.t. adding fences in those locations would "cut off" the error states, and output a program P' with fences added in these locations.

Steps 1 and 2 are standard in the world of software verification (in particular, software model checking [12]). The focus of this work, and the "heart" of our algorithm is therefore step 3. The general approach we use (similarly to the work of Vechev et al. [35]) is not to try to compute fence locations directly, but use an intermediate constraint language.

The high-level idea is that we first choose a constraint language F and associate with every transition t some constraint $\chi(t)$ from F. Very informally, we say that a constraint $\chi(t)$ is *enforceable* if we can "cut the transition t off" from the transition system by adding (syntactic) fences to the program. We call adding such fences *enforcing* the constraint.

We can use this idea to break the problem down into the 3 following major sub-steps: constraint *generation*, *solving* and *implementation*.

1. (Generation) Compute a boolean formula ψ over the constraints that represents *all* of the ways to cut off all error states in E (that is, make them unreachable in the transition system).
2. (Solving) Find a minimal satisfying assignment to ψ. This gives us a minimal constraint set δ that, if enforced, will cut off all error states. Note that it is possible that there are several such minimal constraint sets.
3. (Implementation) Transform δ into a fence placement that enforces those constraints. Again, there may be many ways to implement the constraints as fences.

For this scheme to work, we need to compute ψ such that every satisfying assignment (constraint set) δ of ψ satisfies the following two properties:

- Every constraint in the set δ are enforceable using memory fences inserted into the program code of P.
- If all constraints in δ are enforced by inserting memory fences into P (creating a new program P'), then P' does not violate the specification S.

Additionally, we want the computed constraint formula ψ to be maximally permissive: a constraint set satisfies the formula *if and only if* enforcing it will make the modified program adhere to the specification. This means a minimal satisfying assignment δ of ψ represents a "globally minimal" constraint set: it is impossible to fix the program by enforcing only a strict subset of δ.

Transition System Construction The first stage of the algorithm is to construct the transition system (TS) for the program. The transition system is a graph that consists of vertices which represent the states that can be reached by running the program and edges that represent state transitions. The notation we use for the transition system of the program P is $\langle \sigma_0, \Sigma_P, T_P \rangle$, where Σ_P is the set of states, T_P is the set of transitions, and σ_0 the initial state. The transitions link a state to all of its possible successor states. Here, we assume the input program P is *finite-state under the memory model*. This means that when P is executed under the memory model, there is only a finite number of reachable states. Note that if a program is finite-state under sequential consistency, this does not imply it is also finite-state under a more relaxed model.

Since our algorithm is designed to work with *operational* memory models, given a program state σ we are able to directly compute its set of successors. This means we can construct the transition system iteratively, using a standard worklist algorithm.

Marking Error States Once the TS is constructed, we can identify a subset of *error states*: the set of states that violate the provided safety specification. It is well known that every LTL safety property ϕ can be expressed as Gp where p is a "past-formula", that is, a formula that only refers to the past of the computation. We further assume that that the specification is given as Gp where p is a state property - for example, an assertion on the values of program variables. This restriction is introduced for two reasons:

1. While our algorithm is sound in the general case, it is no longer necessarily optimal. This is because for a general past-formula, fixing the program may be possible not only by cutting off the error states themselves, but by cutting off some of their predecessors. It is possible that our algorithm can be extended to this case, but we did not explore this possibility in detail.
2. This restriction allows us to check whether a state is an error state immediately after we encounter it during exploration. This allows us not to explore any of the error states' descendants, which improves the algorithm's performance.

As many practically useful safety properties can be expressed as Gp where p is a state property, we believe this to be a reasonable restriction.

Constraints Our goal is to transform an input program P into a new output program P' that satisfies S. At this stage, it is convenient to "abstract away" the two programs and focus purely on transition systems. Given a transition system $\langle \sigma_0, \Sigma_P, T_P \rangle$ under a memory model M, we can identify some transitions as *avoidable* and others as *unavoidable*. A transition $t = \sigma_1 \longrightarrow \sigma_2$ is considered avoidable if it is possible to construct a program P' by adding fences to P s.t. $\langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle$ does not contain a transition that corresponds to t. Since discussing two separate transition systems (for P and P') is cumbersome, we informally refer to this process as *cutting* t from the transition system $\langle \sigma_0, \Sigma_P, T_P \rangle$.

More practically, we need to pick (according to the memory model) a set of constraints such that every such constraint can be *enforced* by adding memory fences to P. We then associate with every transition $\sigma_1 \xrightarrow{t} \sigma_2$ a set of constraints $\chi(t)$, that satisfies the following properties:

- If at least one of the constraints in $\chi(t)$ is enforced, then σ_2 is no longer reachable from σ_1 in P'.
- If none of the constraints are enforced, and σ_1 is reachable in P', then σ_2 is also reachable.

In other words, $\chi(t)$ precisely captures *all the ways* to cut the transition t.

This means a transition is avoidable if and only if $\chi(t)$ is non-empty. We can "lift" this definition from transitions to program traces and states: An avoidable trace is a trace in which *at least one* transition is avoidable, and an avoidable state is a state such that *all* program traces leading to it are avoidable.

Given a transition system and a specification we wish to find a constraint set which would cut all traces leading to error states. One possible approach, in the spirit of previous work by Vechev et al. [35], is to enumerate all (acyclic) traces leading to an error state and try to prevent each trace by enforcing appropriate constraints. However, such enumeration does not scale to practical programs as the number of traces can be exponential in the size of the transition system which is itself exponential in program length. Instead, our algorithm works on a state-by-state basis by assigning an *avoid formula* to each state. The avoid formula of a state captures all the ways to cut that state from the transition system.

Suppose we want to cut the state σ. Let the incoming transitions of σ be $t_1, \ldots, t_k$, with source states $\sigma_1, \ldots, \sigma_k$ respectively. To cut σ, we must make it unreachable through all of its incoming transitions. For each transition t_i, this means either cutting t_i itself or removing the source state σ_i. More concretely, we must either enforce some constraint in $\chi(t_i)$ or recursively find the avoid formula for σ_i and enforce some satisfying assignment of that formula. This is in fact recursive only if the transitions system $\langle \sigma_0, \Sigma_P, T_P \rangle$ is acyclic - if it contains cycles, the avoid formula of σ_i may itself depend on the avoid formula of σ. This suggests that the desired avoid formula for a state is a fixed point of a function that relates the avoid formula of a state to those of its predecessor states.

Constraint Generation Algorithm Now that the definitions are in place, we can present the algorithm used for the constraint formula generation phase. Instead of dealing directly with formulae, we will for convenience present the algorithm in terms of boolean functions ("avoid functions"). We will, however, often abuse notation and identify boolean variables with atomic propositions, and monotone boolean functions with monotone propositional formulae that define those functions. In particular:

- For a function f and an assignment of values to variables δ we will use $\delta \models f$ to mean that $f(\delta) = \mathbf{tt}$.
- For functions f_1, f_2, we use $f_1 \implies f_2$ to mean $f_1 \sqsubseteq f_2$, $f_1 \vee f_2$ to mean $f_1 \sqcup f_2$, etc.

Let $\mathbb{V}$ be a set of variables, representing possible constraints on execution. Let F be the set of monotone boolean functions over $\mathbb{V}$ with the standard order relation (also known as the free distributive lattice over $\mathbb{V}$). Let $\langle \sigma_0, \Sigma_P, T_P \rangle$ be a transition system and σ_0 the initial state. Then a legal *labeling function* is a function $L : \Sigma_P \to F$, such that $L(\sigma_0) = \mathbf{ff}$. Intuitively, the labeling function L attaches an avoid function to a state. We require $L(\sigma_0)$ to always be false as the initial state can never be avoided. For a given transition system

$\langle \sigma_0, \Sigma_P, T_P \rangle$, we denote by Λ_P the set of all legal labeling functions for that transition system.

Given a labeling function L and a state $\sigma \in \Sigma_P$, we define:

$$avoid(L, \sigma) = \bigwedge \{L(\mu) \vee \chi(t) \mid (\mu \xrightarrow{t} \sigma) \in T_P\}$$

This formalizes the previously presented intuition: given a labeling L, to avoid a state σ we must avoid all incoming transitions $\mu \xrightarrow{t} \sigma$, either by cutting t (using $\chi(t)$) or by enforcing $L(\mu)$. In the above definition, we abuse notation by treating $\chi(t)$ not as a subset of $\mathbb{V}$, but rather as the function represented by $\bigvee \{p \mid p \in \chi(t)\}$. We will continue this abuse throughout this paper. Whether χ is treated as a set or a function should always be clear from the context.
Using *avoid* we define an operator $trans : (\Sigma_P \to F) \to (\Sigma_P \to F)$ that updates the labeling to the "next generation" of avoid functions:

$$trans[L] = \lambda\sigma \in \Sigma_P . L(\sigma) \wedge avoid(L, \sigma)$$

If L is legal, then so is $trans(L)$ because:

- $trans[L](\sigma_0) = (L(\sigma_0) \wedge ...) = \mathbf{ff}$
- $avoid(L, \sigma)$ is monotone, as is $L(\sigma)$, and a conjunction of two monotone functions is also monotone.

The algorithm to find the desired labeling function is now very simple: we take the initial labeling function L_0 defined below, and iteratively apply *trans* until a fixed point is reached.

$$L_0 = \lambda\sigma \in \Sigma_P . \begin{cases} \mathbf{ff} & \text{if } \sigma = \sigma_0 \\ \mathbf{tt} & \text{if } \sigma \neq \sigma_0 \end{cases}$$

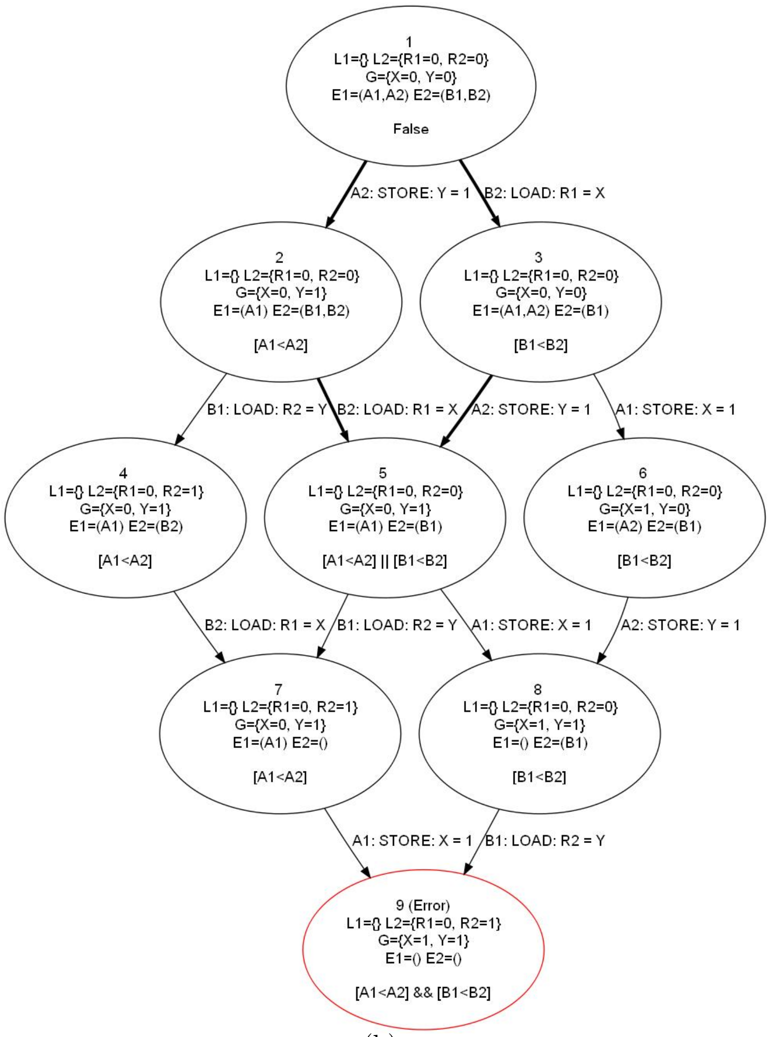
From this point on, we refer to the L function to which the fixed point computation converges as *av*.

However, directly applying this algorithm is inefficient, for two reasons. First, it requires maintaining two copies of the transition system. More importantly, a lot of unnecessary computation is performed because it is possible that in every application of *trans* only few $L(\sigma)$ values actually change. Therefore we use an optimized version based on the standard "chaotic iteration" method due to Cousot & Cousot [13]. This version is shown in Algorithm 1.

Lines 2-4 of the algorithm set the initial labeling to L_0. The labeling is then updated in the following fashion. First, the entire transition system is added to a workset. Then, if the worklist is not empty we pick an arbitrary state σ, and update it from $L(\sigma)$ to $trans[L](\sigma)$ (lines 8 - 10). We then check whether $L(\sigma)$ was changed by the application of *trans*. If it has, we may need to update the labeling of its descendant states, so we add all descendants of σ to the workset. When the workset becomes empty, a fixed point has been reached, so we can return the conjunction of constraints for the error states.

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

(a)



(b)

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

Algorithm 1: Constraint Generation

```
   Input: Program P, Specification S, Memory Model M
   Output: Program P' satisfying S
1  compute $\langle \sigma_0, \Sigma_P, T_P \rangle$ under the memory model M
2  $L(\sigma_0) \leftarrow false$
3  foreach state $\sigma \in \Sigma_P \setminus \{\sigma_0\}$ do
4      $L(\sigma) \leftarrow true$
5  workset$\leftarrow \Sigma_P \setminus \{\sigma_0\}$
6  while workset is not empty do
7      $\sigma \leftarrow$ select and remove state from workset
8      $\varphi \leftarrow L(\sigma)$
9      foreach transition $t = (\mu \longrightarrow \sigma) \in T_P$ do
10         $\varphi \leftarrow \varphi \wedge (L(\mu) \vee \chi(t))$
11     if $L(\sigma) \not\equiv \varphi$ then
12         $L(\sigma) \leftarrow \varphi$
13         add all successors of $\sigma$ in $\Sigma_P$ to workset
14 $\psi \leftarrow \bigwedge\{L(\sigma) \mid \sigma \not\models S\}$
15 return $\psi$.
```

Example Consider the simple concurrent program shown in Fig. 2(a). X and Y are integer variables shared between processes A and B, while R_1 and R_2 are integer variables local to process B. For illustrative purposes, the memory model we use here is a simplified version of RLX (described formally in [18]). In this model any two (data and control) independent instructions can be reordered. However, as opposed to full RLX, stores to shared memory are preformed atomically.

The constraint language we use in the example consists of constraints on execution order. The constraint $[L_1 \prec L_2]$ where L_1 and L_2 are program labels means we forbid L_2 to bypass L_1. That is, if $[L_1 \prec L_2]$ is enforced, and L_1 precedes L_2 in program order, then L_1 must be executed before L_2. We give a more detailed explanation of the constraints later in this section.

Fig. 2(b) shows part of the transition system of this program running on this specific memory model. We only show states that can lead to an error state, as the rest of the transition system is not relevant to the example. Inside each state in the figure we show: (i) assignments to the local variables of each process (L_1 and L_2), and the global variables G; (ii) the execution buffer of each process (E_1 and E_2); (iii) the (final) avoid formula of the state. Since stores are atomic, we do not show B_σ.

For this program, our specification is that $R_1 \geq R_2$ in the program's final state. In the initial state (state 1) all four variables have the value 0. The transition system also contains a single error state (state 9) where $R_1 = 0$ and $R_2 = 1$ (state 9). Since the transition system is acyclic, we can find $av(\sigma)$ by topologically sorting the states, and then computing av once for each state. For example:

- Since state 1 is the initial state, the avoid formula is necessarily **ff**.
- 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 do so we first need to know $\chi(1 \longrightarrow 2)$. Informally, we need to know whether A_2 is executed out of order, and which alternative instructions could have been executed by A instead. If we examine the execution buffer E_1 of state 1 and look at the instructions that precede A_2, we find that A_2 is executed out of order, and that A_1 precedes it in the buffer. This implies we can enforce the constraint $[A_1 \prec A_2]$ as a way to avoid the transition A_2. Since the source state (state 1) cannot be avoided, the avoid formula for state 2 is simply $[A_1 \prec A_2]$. The formula $[B_1 \prec B_2]$ for state 3 is obtained similarly.
- The transition from state 2 to state 4 is taken "in order", that is, it doesn't violate any enforceable constraint. Therefore, the transition itself cannot be avoided and the only way to avoid reaching 4 is through enforcing the avoid formula of its predecessor, state 2. So the avoid formula of state 4 is also $[A_1 \prec A_2]$.
- State 5 has two incoming transitions: B_2 and A_2. B_2 is taken out of order from state 2 and can be prevented by enforcing the constraint $[B_1 \prec B_2]$. The constraint for the source state 2 is $[A_1 \prec A_2]$, so the overall constraint is $[B_1 \prec B_2] \vee [A_1 \prec A_2]$. 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 $[B_1 \prec B_2] \vee [A_1 \prec A_2]$ with itself. In other words, it is $[B_1 \prec B_2] \vee [A_1 \prec A_2]$.
- For the error state 9, the two incoming transitions are executed in-order and cannot be avoided. The overall constraint is thus generated as a conjunction of the constraints of the predecessor states 7 and 8, and it is $[B_1 \prec B_2] \wedge [A_1 \prec A_2]$.

Note that since there is only one error state, the resulting overall formula is the avoid formula of that error state: $[B_1 \prec B_2] \wedge [A_1 \prec A_2]$.

Handling Boolean Functions The algorithm, as presented above, "hides" several representation and performance issues related to boolean functions. The clearest issue is that the algorithm returns a boolean function that represents a constraint formula. However, to actually place fences we require not the formula but rather its minimal satisfying assignments. We "offload" this task to standard SAT solving tools. As our experience shows, the SAT-solving stage is not a performance bottleneck.

A bigger issue is the fact *every* step of the algorithm requires an equivalence check of two boolean functions (the test $L(\sigma) \not\equiv \varphi$ in Line 11). This is NP-hard in general, and remains NP-hard even under the restriction that both functions are monotonic. With an explicit formulae representation those checks become very computationally expensive. However, if the functions are represented as Binary Decision Diagrams (BDDs) [4], then the equivalence check is, in our experience, also not a practical bottleneck.

Algorithm Correctness and Optimality To show our algorithm is correct, we need to demonstrate that (a) the fixed point computation terminates, and (b) once it terminates, enforcing the avoid formula $av(\sigma)$ indeed cuts the state σ. The algorithm we presented above is not only sound, but also maximally permissive. That is, the formula $av(\sigma)$ is the "weakest" (most permissive) formula that describes the constraints that must be enforced to make σ unreachable. The interested reader can find proofs of the above claims in [17].

2.2 Instantiation for a concrete model

In the previous section, we presented a general algorithm for inferring optimal constraints. When we instantiate this algorithm for a specific memory model, we first need to choose the type of constraints we *can* actually enforce. That means our constraints must satisfy at least the following basic property: For every transition t to which we assign $\chi(t) \neq \emptyset$ it is possible use fences to construct a program P' for which the transition system does not contain t.

To choose the constraint language we must first introduce a few more details of the RLX memory model framework. To model the RMM effects, RLX uses "execution buffers" — similar to the "reordering box" of [28] and "local instruction buffer" of [36]. Informally, every processor p processes its instruction stream in its original order. However, "processing" does not in fact mean the instruction is executed. Rather, every instruction is placed in a buffer $E(p)$. An instruction is actually executed when it is removed from the buffer by the environment. If the buffers behave in a FIFO fashion, this is equivalent to sequential consistency. However if the executed instruction is not necessarily the oldest in the buffer, relaxed behavior occurs. In this framework, different RMMs can be specified by providing different rules for removing instructions from the buffer. A complete definition of RLX semantics can be found in [18, 17].

The constraint language appropriate for RLX is the langauge of "ordering constraints" of the form $\psi = [l_1 \prec l_2]$ where l_1, l_2 are program labels. Intuitively, enforcing the constraint $[l_1 \prec l_2]$ means that P' cannot execute the instruction with label l_2 out of order with respect to the instruction at label l_1. We then define $\chi(t)$ for a transition $t = \sigma \longrightarrow \sigma_2$ with label l_t to be $\chi(t) = \{[l \prec l_t] \mid l <_{\sigma,p} l_t\}$. The relation $l <_{\sigma,p} l_t$ holds when:

1. The transition t was caused by executing the instruction at label l_t by some process p.
2. The execution buffer $E(p)$ of state σ contained an instance of the instruction at l before the instruction at l_t.

This is equivalent to saying the transition t represents the instruction at l_t being executed while bypassing l.

To show that the chosen constraint language is useful, we need to show a correspondence between the constraints and syntactic fences. More concretely, we need to show that:

- We know how to enforce any constraint formula produced by the algorithm by adding fences to P.
- Enforcement can be done efficiently. To see this is a non-trivial property, consider the constraint language consisting of a single constraint β, where the enforcement mechanism is "If $\psi = \beta$ add a fence between every two instructions in P". Clearly, we can enforce this constraint, and enforcing it would create a correct program, but this is not the desired outcome.

First, we can show that adding fences can never introduce new error states. It is clear that adding nop operations (with new labels) to a program has no effect on the program's behavior. So it is enough to show that the set of behaviors a program P with a fence at label l has is a subset of the possible behaviors of P with a nop at the same label. This can show this through a simple simulation argument.

Lemma 2.1 *Let P be a program with a* nop *instruction at label l, and P' the program P with the* nop *replaced by a* fence*. Then $\langle \sigma_0, \Sigma_P, T_P \rangle$ simulates $\langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle$.*

This lemma is trivially extendable to replacing any number of nop instructions by fences. After we have established inserting fences cannot add new error states, the next thing we need to show is that we can in fact use syntactic fences to cut any transition t s.t. $\chi(t) \neq \emptyset$. This is established by the following lemma.

Lemma 2.2 *Let P be a program, $t = \sigma_1 \xrightarrow{l_t} \sigma_2$ a transition in $\langle \sigma_0, \Sigma_P, T_P \rangle$ and $v \in \chi(t)$, where $v = [l \prec l_t]$. Let P' be a modification of P s.t. a fence instruction is placed on every control path between l and l_t. Then there is no t' in $\langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle$ that corresponds to t.*

A corollary of the lemma above is that we can enforce any constraint $v = [l_1 \prec l_2]$ (thus cutting any transition t s.t. $v \in \chi(t)$) by placing a fence on every control path between l_1 and l_2. Using these lemmas we can prove the main soundness theorem.

Theorem 2.3 *Let P be a program, S a specification, $\psi = \bigwedge \{av(\sigma) \mid \sigma \nvDash S\}$ and $\delta \models \psi$. Let P' be the program P modified s.t. for any $[l_1 \prec l_2] \in \delta$ a* fence *instruction is placed on every control path between l_1 and l_2. Then $\forall \sigma \in \Sigma_{P'}. \sigma \models S$.*

To show that the produced constraints are optimal, we can prove the following theorem.

Theorem 2.4 *Let P be a program, S a specification, $\psi = \bigwedge \{av(\sigma) \mid \sigma \nvDash S\}$, and P' the program P modified by inserting fences. If for every satisfying assignment $\delta \models \psi$ there exists $[l_1 \prec l_2] \in \delta$ s.t. there is no fence on* any *control path between l_1 and l_2, then there is some $\sigma \in \Sigma_{P'}$ s.t. $\sigma \not\models S$.*

2.3 Synthesizing Fences from Constraints

Theorem 2.3 shows that we can syntactically implement any solution to the constraint formula ψ produced by our algorithm. It shows that if for every constraint $[l_1 \prec l_2]$ that needs to be enforced fences are placed on all control-flow paths between l_1 and l_2 then the resulting program is safe. Unfortunately, while Theorem 2.4 shows a fence must be placed on *some* control path between l_1 and l_2, it does not require placing a fence on *all* of them. There are, in fact, several reasons a fence placement constructed by simply taking some minimal satisfying assignment δ of ψ and adding fences on all control-flow paths may be suboptimal:

- It is not even clear which optimality metric we should use. The number of (static) fences added to the program seems like a convenient choice, but may be misleading. Several fences placed before a loop may have a much smaller (dynamic) execution cost than a single fence placed inside the loop body.

- Theorem 2.4 shows we must add a fence on *some* control path of every constraint that belongs to a minimal satisfying assignment, as opposed to *all* control-flow paths. This is not a weakness of the theorem, as placing fences on all paths is in fact not always required. This may happen for two reasons. First, some control-flow paths may be infeasible, and putting fences on these paths is thus unnecessary. More subtly, it is possible that a given re-ordering of instructions is only harmful on some execution paths. Our chosen constraint language does not preserve enough information to make these distinctions. We could use an alternative constraint language to preserve it, but this would dramatically increase the size of Λ_P — the number of possible constraints would be exponential in the number of labels, as opposed to quadratic.

- Often, it is possible to satisfy several constraints with a single fence. Thus a judicious placement of fences is still required, even once a minimal assignment to the constraint formula is known. Moreover, different minimal assignments may lead to different placement tradeoffs.

We resolve the first issue by working with the natural partial order on fence placements: a set of added fences C is better then a set C' if $C' \subseteq C$. We then produce all minimal incomparable placements and leave the choice between them to the programmer. Choosing between incomparable (by containment) fence placements is a separate hard problem, which we leave to future work.

The second issue could be resolved by adopting a more precise "flow-sensitive" constraint language. This could be done by encoding in the constraints $\chi(t)$ of a transition $t = \sigma_1 \longrightarrow \sigma_2$ information about program paths that lead to t. Moreover, if we used a "context-sensitive" implementation mechanism instead of fences (for example "conditional fences" — fences that are only sometimes executed, depending on the current program state) we could use even finer constraints. For the input programs we used, none of these improvements were necessary. Therefore, we also defer examination of these alternatives to the future.

The third issue requires further examination. While there are in general many ways to implement a given constraint $v = [l_1 \prec l_2]$, for simple programs it usually sufficient (while clearly not optimal in general) to consider two options:

- Place a fence immediately after instruction l_1
- Place a fence immediately before instruction l_2 (if there are branch instructions pointing to l_2 they should point to the newly added fence).

This is complicated slightly by the fact that even in this case, there is interdependence between constraints. For example, consider a program with three statements with labels l_1, l_2, l_3 in sequence and the constraint formula $v_1 \wedge v_2$ where $v_1 = [l_1 \prec l_2], v_2 = [l_1 \prec l_3]$. Obtaining the (only) solution $\{v_1, v_2\}$ and then deciding to place a fence immediately before l_2 (to enforce v_1) and before l_3 (to enforce v_2) will result in a placement that contains two fences, instead of the expected single fence after l_1. We solve this by replacing the constraint formula ψ with a new formula ζ.

A fence may only be placed after an existing code label. Therefore, for each label we define a new variable v_l. We also define a function *prev* that returns for each label l the preceding (in the program code) label. We then produce ζ by replacing every variable $v = [l_1 \prec l_2]$ in ψ with the clause $v_{l_1} \vee v_{prev(l_2)}$. It is easy to show that every satisfying assignment to ζ still produces a sound fence placement. However, it alleviates the interdependence problem by "off-loading" it to a SAT solver. In the preceding example, the formula $v_1 \wedge v_2$ is transformed into $v_{l_1} \wedge (v_{l_1} \vee v_{l_2})$, with the minimal solution $\{v_{l_1}\}$ as desired.

Limitations The main drawback of the algorithm described in this section is the fact that it requires explicit enumeration of the program's state-space. While this is possible for some programs, many programs for which we want to infer fences do not allow such explicit enumeration because the state-space is not finite. This might happen due to a combination of several reasons. Three common reasons are:

1. We are interested in inferring a fence placement for an open system (e.g. library code), and not a single finite-state program.
2. The program for which we wish to infer fences utilizes a potentially unbounded number of heap locations.
3. The program is finite-state under SC but not finite-state under the desired relaxed memory model.

In case (1), the problem boils down to the fact we are not interested in placing fences in a single program. Rather, we want to place fences in the code of a library *implementation* such that it remains correct irrespective of the code *using* the library (the data structure *client*). A different way to phrase this is to say we want the *most general* client (which represents all possible clients) of the library to be correct. Unfortunately, the most general client itself is usually not a finite-state program. For example, consider a queue implementation that uses a linked list as the underlying data representation. A client that may add an unbounded number of elements to the queue will use unbounded memory, and the state-space for the client/queue combination is unbounded. In general, this is a hard problem that we do not try to completely solve. We attempt to reduce it using two methods: (a) Hand-picked clients that we believe are representative of the data structure's behavior. (b) Exhaustive enumeration of clients up to a specified bound on the number of operations. Neither of these two solutions produces a sound verification (or fence inference) procedure. However, in practice these methods allow us to infer optimal fences for realistic data structures. We have verified that the results are indeed optimal by manually comparing them to fence placements found in the literature.

In case (2), the problem is that due to use of an unbounded number of heap locations, the program is infinite-state even under the sequentially-consistent model. One way to deal with this problem is to "work around it" by applying the algorithm to slightly different programs, and dealing with the difference separately (e.g. using finite-state clients instead of the infinite-state most general client as in case (1)). Another is to use heap abstractions.

Regarding case (3), as Atig et al. have shown [2], this is in general a very hard problem. Given a finite-state (under SC) program P, deciding reachability for the same program under SPARC TSO or PSO has non-primitive recursive complexity. Further, under SPARC RMO, reachability for SC-finite-state programs becomes undecidable. One way to deal with

```
void enqueue(queue_t *queue, value_t value)
{
    node_t *node, *tail, *next;
    node = new_node();
    node->value = value;
    node->next = 0;
    fence("store-store");
    while (true) {
        tail = queue->tail;
        fence("load-load");
        next = tail->next;
        fence("load-load");
        if (tail == queue->tail)
        if (next == 0) {
        if (cas(&tail->next,
            (unsigned) next, (unsigned) node))
            break;
        } else
            cas(&queue->tail,
                (unsigned) tail, (unsigned) next);
    }
    fence("store-store");
    cas(&queue->tail,
        (unsigned) tail, (unsigned) node);
}
```

Figure 3: Enqueue operation of the Michael-Scott queue (from [6])

problems of this kind is through the use of abstract interpretation — a technique explored in [19] and [17].

3 Experimental Evaluation

We have implemented our algorithm in a pair of tools called FENDER and BLENDER. FENDER is a direct implementation of the fence inference algorithm of section 2 for the RLX framework. In BLENDER we adapted the implementation to work with a wider range of memory models [19]. To give a flavor of the capabilities of these tools, we present the fence inference results for the Michael-Scott nonblocking queue [27]. This queue is one of few algorithms for which a correct fence placement (for RMO) has been previously published [6]. We refer to that placement of fences as the "reference placement". The reference placement uses 7 fences, 4 in `enqueue()`, and 3 in `dequeue()`. As [6] notes, all of the fences were found using small test-cases. Our hypothesis was that by running FENDER with a small number of test-cases, we can automatically infer the appropriate fences.

Under RMO (which is closest to the model used by [6]), a small set of clients produced 20 different sets of 4 constraints. Using the local fence placement method there are only 4 different ways to implement those sets using 3 fences: 1 fence in `enqueue()` and 2 in `dequeue()`. One of those placements was, as expected, a proper subset of the reference placement found in [6], and the others were similar.

Fig. 3 is copied verbatim from [6] and shows the `enqueue()` method for the algorithm (including 4 of the 7 fences placed using CheckFence). The reference placement contains 7 fences, while our tool inferred only 3 of these 7, which may seem, at first glance, insufficient. However, manual examination of the 4 missing fences confirms that they are in fact redundant in our model.

- The load-load fence on line 10 of Fig. 3 prevents the load on line 11 from being executed before the load on line 9 (note that the two loads are data dependent). To the best of our understanding SPARC RMO only allows control speculation, but not data speculation, which means this fence is in fact not necessary.
- The store-store fence on line 22 prevents the CAS on line 23 from being executed before the CAS on line 15. However, this may only happen if the CAS on line 23 is executed speculatively, since its execution is control-dependent on the *success* of the CAS in line 15. Under RMO operations that write to memory (and, in particular, CAS operations) may not be executed speculatively, so this fence is never needed.
- The two load-load fences on line 12 and on line 57 of the `dequeue()` code given in [6] enforce the correct execution of a construct meant to solve a certain type of ABA problem that only occurs when immediate reuse of memory is allowed. However, under the assumption of automatic memory management, the statements in lines 13 and 58 are redundant (see [25]). Since the correct execution of these two statements is no longer important, FENDERcorrectly omits the two fences that "protect" them.

Under PSO, only sets of two constraints (implementable by a single store-store fence) in the `enqueue()` method were inferred. This is consistent with the fact that, under this model, loads are not reordered with each other so load-load constraints are unnecessary. Under TSO, no fences were inferred, again consistent with our expectations, and with the claim in [6] that under the x86 memory model (which resembles TSO), no fences should be necessary.

A wide range of our experimental results, as well as details on our methodology, appear in [18, 19].

4 Related Work

Several automated techniques to place memory fences in concurrent programs have been developed over the years. A large body of work dating back to the late 1980s relies on the concepts of *delay set analysis* of Shasha & Snir [29] for reasoning about relaxed memory models. This analysis enables one to find all potential conflicts (more or less equivalent to data races), and place fences accordingly. A fence inference scheme based on delay set analysis was successfully implemented in the "Pensieve" Java compiler [21, 14, 31], which can effectively process large amounts of code. However, a violation of SC does not necessarily cause a violation of any high-level properties. Thus those algorithms are often needlessly conservative. Unlike this previous work, the approach outlined in this paper, uses a high-level specification and allows a trade-off between performance and optimality of the solution.

Another possible approach to fence inference is to use a verification tool combined with syntactic exploration. There exist several techniques for program testing and verification under relaxed memory models, and tools have been developed that implement these techniques (cf. [6, 7, 9, 8, 22].) To utilize a verification tool (e.g. CheckFence [6]) for inference, the programmer may use 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 verification 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. In [23], Linden & Wolper automate this approach, using the technique described in [22] as the underlying verification tool. However, their tool still suffers from the same problem - it does not necessarily provide a globally optimal solution.

5 Conclusion

We presented a novel fence inference algorithm and demonstrated its practical effectiveness by evaluating it on various challenging state-of-the-art concurrent algorithms. The work presented here is a small sample from our wider work on *synthesis of synchronization* in concurrent programs (e.g., [32, 34, 33, 35, 24]). In the future, we plan to extend our techniques to handle infinite-state programs (e.g., heap-manipulating programs) running on relaxed memory models.

Acknowledgements

Eran Yahav is a Deloro Fellow. This research was partially supported by The Israeli Science Foundation (grant no. 965/10).

References

[1] Adve, S. V., and Gharachorloo, K. Shared memory consistency models: A tutorial. *IEEE Computer 29* (1995), 66–76.

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

[3] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: an efficient multithreaded runtime system. In *PPOPP '95*.

[4] Bryant, R. E. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv. 24*, 3 (1992), 293–318.

[5] Burckhardt, S., Alur, R., and Martin, M. M. K. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *CAV'06*.

[6] Burckhardt, S., Alur, R., and Martin, M. M. K. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI* (2007), pp. 12–21.

[7] Burckhardt, S., and Musuvathi, M. Effective program verification for relaxed memory models. In *CAV* (2008), pp. 107–120.

[8] Burnim, J., Sen, K., and Stergiou, C. Testing concurrent programs on relaxed memory models. In *ISSTA '11*, pp. 122–132.

[9] Burnim, J., Sen, K., and Stergiou, C. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS'11* (2011), pp. 11–25.

[10] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA* (2005), pp. 519–538.

[11] Chase, D., and Lev, Y. Dynamic circular work-stealing deque. In *SPAA* (2005), pp. 21–28.

[12] Clarke, E. M., Grumberg, O., and Peled, D. *Model Checking.* The MIT Press, 1999.

[13] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL* (1977), pp. 238–252.

[14] Fang, X., Lee, J., and Midkiff, S. P. Automatic fence insertion for shared memory multiprocessing. In *ICS* (2003), pp. 285–294.

[15] Gharachorloo, K., Gupta, A., and Hennessy, J. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *ASPLOS'91.*

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

[17] Kuperstein, M. Preserving correctness under relaxed memory models. Master's thesis, Technion, 2012.

[18] Kuperstein, M., Vechev, M., and Yahav, E. Automatic inference of memory fences. In *FMCAD* (2010), pp. 111–119.

[19] Kuperstein, M., Vechev, M., and Yahav, E. Partial-coherence abstractions for relaxed memory models. In *PLDI '11* (2011), pp. 187–198.

[20] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput. 28*, 9 (1979), 690–691.

[21] Lee, J., and Padua, D. A. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput. 50*, 8 (2001), 824–833.

[22] Linden, A., and Wolper, P. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN* (2010), pp. 212–226.

[23] Linden, A., and Wolper, P. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN* (2011), pp. 144–160.

[24] Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., and Yahav, E. Dynamic synthesis for relaxed memory models. PLDI '12, to appear.

[25] Michael, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC* (2002), pp. 21–30.

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

[27] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC* (1996), pp. 267–275.

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

[29] Shasha, D., and Snir, M. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst. 10*, 2 (1988), 282–312.

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

[31] Sura, Z., Wong, C.-L., Fang, X., Lee, J., Midkiff, S. P., and Padua, D. A. Automatic implementation of programming language consistency models. *LNCS 2481* (2005), 172.

[32] Vechev, M., and Yahav, E. Deriving linearizable fine-grained concurrent objects. In *PLDI* (2008), pp. 125–135.

[33] Vechev, M., Yahav, E., Bacon, D. F., and Rinetzky, N. CGCExplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI* (2007), pp. 456–467.

[34] Vechev, M., Yahav, E., and Yorsh, G. Inferring synchronization under limited observability. In *TACAS* (2009), pp. 139–154.

[35] Vechev, M., Yahav, E., and Yorsh, G. Abstraction-guided synthesis of synchronization. In *POPL '10* (2010).

[36] Yang, Y., Gopalakrishnan, G., and Lindstrom, G. UMM: an operational memory model specification framework with integrated model checking capability. *Concurr. Comput. : Pract. Exper. 17*, 5-6 (2005), 465–487.