Dynamic Synthesis for Relaxed Memory Models

Feng Liu
Princeton University
fengliu@princeton.edu

Nayden Nedev
Princeton University
nnedev@princeton.edu

Nedyalko Prisadnikov
Sofia University
nedy88@gmail.com

Martin Vechev
ETH Zürich
martin.vechev@inf.ethz.ch

Eran Yahav *
Technion
yahave@cs.technion.ac.il

Abstract

Modern architectures implement relaxed memory models which may reorder memory operations or execute them non-atomically. Special instructions called *memory fences* are provided, allowing control of this behavior.

To implement a concurrent algorithm for a modern architecture, the programmer is forced to manually reason about subtle relaxed behaviors and figure out ways to control these behaviors by adding fences to the program. Not only is this process time consuming and error-prone, but it has to be repeated every time the implementation is ported to a different architecture.

In this paper, we present the first scalable framework for handling real-world concurrent algorithms running on relaxed architectures. Given a concurrent C program, a safety specification, and a description of the memory model, our framework tests the program on the memory model to expose violations of the specification, and synthesizes a set of necessary ordering constraints that prevent these violations. The ordering constraints are then realized as additional fences in the program.

We implemented our approach in a tool called DFENCE based on LLVM and used it to infer fences in a number of concurrent algorithms. Using DFENCE, we perform the first in-depth study of the interaction between fences in real-world concurrent C programs, correctness criteria such as sequential consistency and linearizability, and memory models such as TSO and PSO, yielding many interesting observations. We believe that this is the first tool that can handle programs at the scale and complexity of a lock-free memory allocator.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]

General Terms Algorithms, Verification

Keywords Concurrency, Synthesis, Relaxed Memory Models, Weak Memory Models

1. Introduction

Modern architectures use relaxed memory models in which memory operations may be reordered and executed non-atomically [2]. These models enable improved hardware performance but pose a burden on the programmer, forcing her to understand the effect that the memory model has on their implementation. To allow programmer control over those executions, processors provide special *memory fence* instructions.

As multicore processors increasingly dominate the market, highly-concurrent algorithms become critical components of many systems [28]. Highly-concurrent algorithms are notoriously hard to get right [22] and often rely on subtle ordering of events, which may be violated under relaxed memory models [14, Ch.7].

Placing Memory Fences Manually reasoning where to place fences in a concurrent program running on a relaxed architecture is a challenging task. Using too many fences (over-fencing) hinders performance, while missing necessary fences (under-fencing) permits illegal executions. Manually balancing between over- and under- fencing is very difficult, time-consuming and error-prone [4, 14]. Furthermore, the process of placing fences is repeated whenever the algorithm changes, and whenever it is ported to a different architecture.

Our goal is to automate the task of fence placement and free the programmer to focus on the algorithmic details of her work. Automatic fence placement is important for expert designers of concurrent algorithms, as it lets the designer quickly prototype with different algorithmic choices. Automatic fence placement is also important for any programmer trying to implement a concurrent algorithm as published in the literature. Because fence placement depend on the specific architecture, concurrent algorithms are usually published without a detailed fence placements for different architectures. This presents a nontrivial challenge to any programmer trying to implement an algorithm from the literature on relaxed architectures.

Dynamic Synthesis Existing approaches to automatic fence inference are either overly conservative [26], resulting in over-fencing, or have severely limited scalability [17, 18]. The main idea in this paper is to break the scalability barrier of static approaches by performing the synthesis based on *dynamic executions*. To identify illegal executions, we introduce a *flush-delaying* demonic scheduler that is effective in exposing illegal executions under relaxed memory models.

Given a program P, a specification S, and a description of the memory model M, guided execution (using the flush-based scheduler) of P under M can be used to identify a set of illegal execu-

[*] Deloro Fellow

PLDI'12, June 11–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/06...$10.00

```
1   void put(int task) {
2    t = T;
3    items[t] = task;
4    fence(st-st); //F2
5    T = t + 1;
6    fence(st-st); //F3
7   }
```

```
1   int steal() {
2    while (true) {
3     h = H;
4     t = T;
5     if (h >= t)
6      return EMPTY;
7     task = items[h];
8     if(!cas32(&H,h,h+1))
9      continue;
10   return task;
11   }
12  }
```

```
1   int take() {
2    while (true) {
3     t = T - 1;
4     T = t;
5     fence(st-ld); //F1
6     h = H;
7     if (t < h) {
8      T = h;
9      return EMPTY;
10    }
11    task = items[t];
12    if (t > h)
13     return task;
14    T = h + 1;
15    if(!cas(&H,h,h+1))
16     continue;
17    return task;
18   }
19  }
```

Figure 1: Simplified version of the Chase-Lev work-stealing queue. Here, store-load fence F1 prevents the non-SC scenario of Fig. 2a under TSO; F1 combined with store-store fence F2 prevents the non-SC scenario of Fig. 2b under PSO. F1, F2 and store-store fence F3 prevent the linearizability violation of Fig. 2c under PSO.

tions (violating S). The tool will then automatically synthesize a program P' that avoids the observed illegal executions (under M), but still permits as many legal executions as possible.

Evaluation Enabled by our tool, we perform the first in-depth study of the subtle interaction between: i) required fences in a number of real-world concurrent C algorithms (including a lock-free memory allocator), ii) correctness criteria such as linearizabiltiy and (operation-level) sequential consistency [14], and iii) relaxed memory models such as TSO and PSO.

Main Contributions The main contributions of this paper are:

- A novel framework for dynamic synthesis of synchronization under relaxed memory models. The framework breaks the scalability barrier of static synthesis approaches by performing the synthesis based on dynamic executions.
- A flush-delaying demonic scheduler which delays the effect of global stores on shared memory and attempts to expose illegal executions under relaxed memory models.
- An implementation of our tool in LLVM. The implementation required a number of extensions of the LLVM interpreter (e.g. adding support for multi-threading) which are useful for general concurrency analysis and for plugging-in new memory models.
- An evaluation on a range of real world concurrent C algorithms, including work-stealing queues and a lock-free memory allocator. We believe this to be the first tool that can handle algorithms at the scale of a lock-free memory allocator.
- An in-depth study on the connection between: i) relaxed memory models such as TSO and PSO, ii) required fences in practical concurrent algorithms, and iii) correctness criteria such as linearizabiltiy and (operation-level) sequential consistency. Our results yield a number of insights, useful in understanding the interplay between relaxed memory models and concurrency.

2. Overview

In this section, we show how different synchronization requirements arise for different memory models and specifications. We use a practical work-stealing queue algorithm as an example.

Motivating Example Fig. 1 shows a pseudo-code of the Chase-Lev work-stealing queue [7] (note that our implementation handle the complete C code). A work-stealing queue is a special kind of double-ended queue that provides three operations: put, take, and steal. A single owner thread can put and take an item from the tail of the queue, and multiple thief threads can steal an item from the head of the queue.

In the implementation of Fig. 1, H and T are global shared variables storing head and tail indices of the valid section of the array items in the queue. The operations put and take operate on one end of the array, and steal operates on the other end.

The put operation takes a task as parameter, and adds it to the tail of the queue by storing it in items[t] and incrementing the tail index T. The take operation uses optimistic synchronization, repeatedly trying to remove an item from the tail of the queue, potentially restarting if the tail and the head refer to the same item. take works by first decrementing the tail index, and comparing the new value with the head index. There are three possible cases:

- new tail index is smaller than head index, meaning that the queue is empty. In this case, the original value of tail index is restored and the take returns EMPTY (line 9).
- new tail index is larger than head index, take then uses it to read and return the item from the array.
- new tail index equals to head index, in which case, the only item in the queue may be potentially stolen by a concurrent steal operation. A compare-and-swap (CAS) instruction is used to check whether the head has changed since we read it into h. If the head has not changed, then there is no concurrent steal, and the single item in the queue can be returned, while the head is updated to the new value h+1. If the value of head has changed, take restarts by going to the next loop iteration.

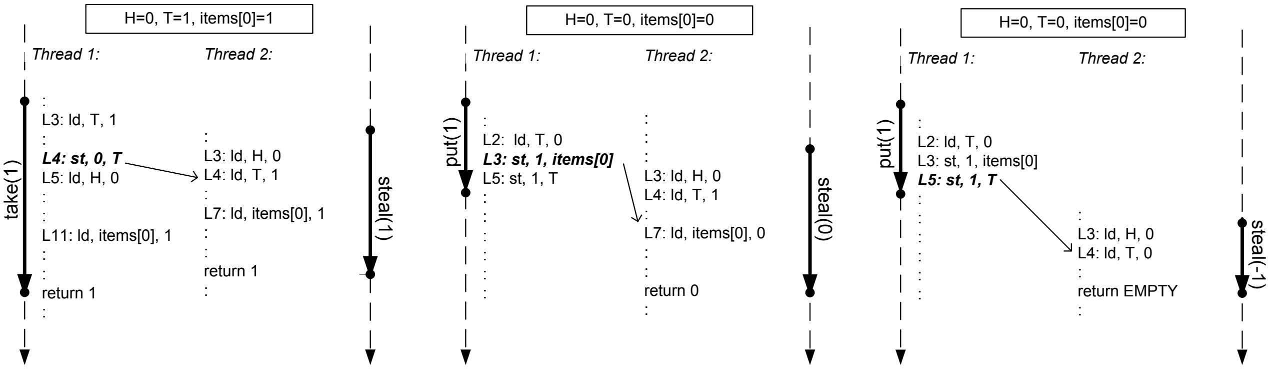
Similarly, the implementation of steal reads the head and tail indexes of the array first, and if the head index is larger or equal to the tail index, either the queue is empty or the only item is taken by the owner, and the thief returns EMPTY. Otherwise, it can read the items pointed by the head. In case no other thieves are stealing the same element, a CAS instruction is used to update the head index atomically. If the CAS succeeds, the item can be returned, otherwise, the steal function needs to retry.

Correctness Criteria and Terminology The implementation of Fig. 1 works correctly on an ideal processor without memory model effects. Under this ideal processor, the two CAS instructions in the algorithm are sufficient for correctly updating the head index H.

In this work, we focus on two important specifications: (operation-level) sequential consistency and linearizability. Note that the term operation-level sequential consistency refers to the behavior of the algorithm and *not* to the memory model itself. Precise definitions and extensive discussion comparing (operation-level) sequential consistency and linearizability can be found in [14, Ch. 3.4-3.5]. As we focus on relaxed memory models, in the rest of the paper we use the term sequential consistency to mean operation-level sequential consistency (and not hardware-level sequential consistency). Informally, both, (operation-level) sequential consistency and linearizability can be expressed as follows:

- Sequential consistency requires that for each concurrent execution, there exists a serial execution that yields the same result.
- Linearizability requires that for each concurrent execution, there exists a serial execution that yields the same result and maintains the ordering between non-overlapping operations in the concurrent execution.

We believe this work is the first detailed comparison between linearizability and sequential consistency specifications for real concurrent algorithms.



(a) Sequential Consistency Violation on TSO

(b) Sequential Consistency Violation on PSO

(c) Linearizability Violation on TSO/PSO. Note that the execution satisfies Sequential Consistency

Figure 2: Executions of a work-stealing queue violating different specifications under different memory models. Arguments of `take` and `steal` are the return values of the operations.

Returning to our algorithm, unfortunately, a relaxed memory model may reorder accesses to shared variables, leading to unexpected results. For example, in the total store order (TSO) memory model, loads can bypass earlier stores in the same thread, thus another thread can load an inconsistent value and produce an incorrect result. In the partial store order (PSO) memory model, both loads and stores can bypass earlier stores.

Correctness under TSO Fig. 2a shows an execution of the Chase-Lev queue under the TSO memory model violating sequential consistency. In this execution, thread 1 performs a `take(1)` operation, and thread 2 performs `steal(1)`, trying to steal the same item. Thread 1 first updates the tail index of the queue to 0, and then loads 1 from the head index. For thread 1, the head equals to the tail, and it takes the first element of the queue, 1, successfully. However, since the update of the tail index by thread 1 is buffered (noted in boldface), thread 2 can only load the original 1 from the tail index, and returns the same item 1. The same element is popped by two different threads, and we cannot find a serial execution that has the same result. The original implementation violates the sequential consistency on TSO memory model. To fix this problem, we need to guarantee that the buffered store of the tail index in thread 1 is committed to main memory and visible to other threads before the head index is loaded. This can be accomplished by inserting a memory fence after the store (F1 in Fig. 1), which enforces a flush of the buffered store to main memory.

Correctness under PSO Unfortunately, fence F1 is not sufficient when running the algorithm on the PSO memory model. Under PSO, a possible store-store reordering can still lead to violation of sequential consistency as shown in Fig. 2b. In this execution, thread 1 performs `put(1)` and thread 2 performs `steal(1)`. Thread 1 tries to store the item 1 in the first position of the queue and updates the tail index in main memory. Concurrently, thread 2 loads the new tail index, finds there is a single item in the queue and loads the item. However, the real value of the item is buffered by thread 1, and thread 2 *reads an uninitialized value* 0. For this execution, we cannot find a serial execution that can fetch an item which has not been put by any of the threads. To fix this problem, before thread 1 updates the tail index, we need to ensure that the corresponding item is updated first. Thus a memory fence F2 is added after the item store instruction. In general, this store-store reordering can also lead to a memory safety violation if for instance we use the value to index some array.

$load(x, r)$	load contents of global variable x into local variable r.
$store(r, x)$	store value of local variable r into global variable x.
$cas(x, r, s, q)$	an atomic compare-and-swap action. If the value in global variable x is as same as that in local variable r, then store in x the value in local variable s and set local variable q to $true$. Otherwise, do nothing and set q to $false$.
$call(f, \vec{a})$	function call of f with argument list $\vec{a}$.
$return(f, r)$	return from a function f with value r.
$fork(t, l)$	create a new thread t to execute statement starting at label l.
$join(t, u)$	thread t joins to thread u.
$fence$	memory fence instruction.
$self()$	return the id of the calling thread.
$flush(x)$	flush the value of shared variable x to main memory (specific to a given memory model, defined later). This statement is inserted by the scheduler, and we use it to model the relaxed memory model effect.

Table 1: Basic statements of our language.

Linearizability Linearizability is stricter than sequential consistency. Linearizability requires at least one additional fence in this algorithm.

Fig. 2c shows an execution of the implementation that satisfies SC under PSO (Fig. 2b), and shows that it violates linearizability. The schedule is similar to the one of Fig. 2b: thread 1 executes `put(1)`, and thread 2 executes `steal(-1)` after thread 1 finishes `put(1)`. Even though the store to the first item by thread 1 is committed to main memory before the tail index is updated, thread 2 can load an incorrect value of the tail index due to the buffering in thread 1, resulting in an empty steal. Because `put(1)` of thread 1 and `steal(-1)` of thread 2 are non-overlapping, we cannot find a serial execution where the thief fails to steal from a queue that is non-empty. The violation occurs because the update to tail is not committed to main memory. To flush the value of the tail index to main memory, another fence (F3 in Fig. 1) needs to be inserted before the method returns. This violation can also occur under TSO, and a memory fence is required there as well.

3. Language

In this section, we present the syntax of our language and the semantics of the two memory models used in the paper: Total Store Order (TSO) and Partial Store Order (PSO).

Syntax We consider a basic concurrent programming language augmented with procedure calls, where the basic statements are listed in Tab. 1. We denote the set of statements by $Stmt$. Given a program, we use $Label$ to denote the set of program labels. We also use the special labels $Inactive$ and $Finished$ to denote that a thread has not yet started or has completed execution. Statements in the program are uniquely labeled and for a label $l \in Label$, we use $stmt(l)$ to denote its corresponding statement in the program.

Semantics 1 Operational semantics under PSO.

$$\frac{stmt(pc) = load(x, r) \qquad B(x) = \epsilon \qquad G(x) = v}{L'(r) = v \qquad pc' = n(pc)} \ \text{(LOAD-G)}$$

$$\frac{stmt(pc) = load(x, r) \qquad B(x) = b \cdot v}{L'(r) = v \qquad pc' = n(pc)} \ \text{(LOAD-B)}$$

$$\frac{stmt(pc) = store(r, x) \qquad B(x) = b \qquad L(r) = v}{B'(x) = b \cdot v \qquad pc' = n(pc)} \ \text{(STORE)}$$

$$\frac{B(x) = v \cdot b}{B'(x) = b \qquad G'(x) = v} \ \text{(FLUSH)}$$

$$\frac{stmt(pc) = fence \qquad \forall x.B(x) = \epsilon}{pc' = n(pc)} \ \text{(FENCE)}$$

$$\frac{\begin{array}{c} stmt(pc) = cas(x, r, s, q) \\ G(x) = L(r) \qquad L(s) = v \qquad B(x) = \epsilon \end{array}}{G'(x) = v \qquad L'(q) = true \qquad pc' = n(pc)} \ \text{(CAS-T)}$$

$$\frac{stmt(pc) = cas(x, r, s, q) \qquad G(x) \neq L(r) \qquad B(x) = \epsilon}{L'(q) = false \qquad pc' = n(pc)} \ \text{(CAS-F)}$$

$$\frac{stmt(pc) = fork(t, l) \qquad pc(t) = Inactive}{\forall x.B(t, x) = \epsilon \qquad pc(t) = l \qquad \forall l.L(t, l) = 0 \qquad pc' = n(pc)} \ \text{(FORK)}$$

$$\frac{\begin{array}{c} stmt(pc) = join(t, u) \\ pc(u) = Finished \qquad \forall x.B(u, x) = \epsilon \end{array}}{pc' = n(pc)} \ \text{(JOIN)}$$

Semantics We use $Thread$ to denote a finite set of thread identifiers and $tid \in Thread$ to denote a thread identifier. A transition system is a tuple $\langle s_0, \Sigma, T \rangle$, where Σ is the set of program states, $s_0 \in \Sigma$ is the initial program state, and T is the transition relation: $T \subseteq \Sigma \times Thread \times Stmt \times \Sigma$.

A transition $s \xrightarrow{tid\,:\,stmt} s' \in T$ holds if $s, s' \in \Sigma$ and if executing statement $stmt$ in state s by thread tid results in state s'. For a transition t, we use $tr(t)$ to denote its executing thread, $label(t)$ for the label whose statement was executed, and $stmt(t)$ (overloaded) to obtain the statement.

A finite execution π is a sequence of transitions starting from the initial state:

$$\pi = s_0 \xrightarrow{tid_1\,:\,stmt_1} s_1 \xrightarrow{tid_2\,:\,stmt_2} \ldots \xrightarrow{tid_n\,:\,stmt_n} s_n.$$

A program state s is defined as a tuple $\langle pc, L, G \rangle$, where:

- $pc \in PC$, where $PC = Thread \rightarrow Label$ is a map of threads to program labels.
- $L \in Env$ where $Env = Thread \rightarrow Local \rightarrow D$ is a map from threads to local variables to values.
- $G \in GVar$ where $GVar = Shared \rightarrow D$ is a map from global shared variables to values.

Here D represents the domain from which the program variables take values. When updating mappings, we use $M'(x) = v$ as a shorthand for $M' = M[x \mapsto v]$. Given a function $pc \in PC$, if the thread tid is clear from the context, we use pc for $pc(tid)$, $stmt(pc)$ to denote the statement at pc, and $n(pc)$ to denote the statement following pc.

Semantics under Memory Models The full small-step (interleaving) transition semantics on sequentially consistent machines is standard. Here, we only give semantics for the statements affected by the memory model: TSO and PSO. To accomplish that, we augment the program state with a store buffer B, which is used to model the memory model effects:

- PSO: $B \in Thread \rightarrow Shared \rightarrow D^*$. That is, we keep a buffer of values for each (thread, shared variable) pair.
- TSO: $B \in Thread \rightarrow (Shared \times D)^*$. That is, we keep a single buffer of values for all variables in a thread.

The semantics of the relevant statements for PSO are given in Semantics 1. The semantics of TSO are the same except that we work with the per-thread buffer, and so they are more restrictive. For instance, in the global load rule LOAD-G, instead of checking that $B(x) = \epsilon$, we will check that $\forall (y, d) \in B.y \neq x$.

Given a program P and memory model M, we use $[\![P]\!]_M$ to denote the set of all possible program executions starting from the initial state.

4. Dynamic Synthesis

In this section, we discuss the algorithm for dynamically synthesizing memory fences. This algorithm can be combined with any demonic scheduler that tries to expose violating executions. The basic idea is to repair the program iteratively: whenever bad executions are encountered, they are repaired so that new executions cannot repeat the same errors.

The high level dynamic synthesis approach is shown in Algorithm 1. The input to the algorithm is a concurrent program P, a memory model M and a specification S. First, the algorithm attempts to trigger an execution that violates the specification. If such an execution is found, then the function $avoid$ computes all possible ways to avoid the execution and adds that to the formula φ. Next, we have a non-deterministic choice "*?*" between enforcing the possible solutions from φ into the program or choosing to accumulate repairs from more violating executions. In practice, "*?*" can be determined in various ways: user-specified bound, dynamically computed value, a heuristic, and so on. When the algorithm finds no violating execution, it aborts with the program P' (and if necessary, enforces any outstanding constraints left in φ). Next, we elaborate on the details of $avoid$ and $enforce$.

Algorithm 1: Dynamic Synchronization Synthesis

```
Input: Program $P$, Specification $S$, Memory Model $M$
Output: Program $P'$ with restricted synchronization
$P' = P$;
$\varphi = true$;
while $true$ do
    select $\pi \in [\![P']\!]_M$ such that $\pi \not\models S$
    if no such $\pi$ exists then
        return $enforce(\varphi, P')$
    $\delta = avoid(\pi, M)$
    if $\delta$ = false then
        abort "cannot be fixed"
    $\varphi = \varphi \wedge \delta$
    if ? then
        $P' = enforce(\varphi, P')$
        $\varphi = true$;
```

Semantics 2 Instrumented semantics under PSO.

$$\frac{stmt(pc) = load(x, r) \qquad \psi = v}{\psi' = v \vee \bigvee\{[l_y \prec pc] \mid y \neq x \wedge B^{\natural}(y) \neq \epsilon \wedge l_y \in B^{\natural}(y)\}} \ \text{(LOAD-G)}$$

$$\frac{stmt(pc) = store(r, x) \qquad B^{\natural}(x) = l \qquad \psi = v}{\begin{array}{c}\psi' = v \vee \bigvee\{[l_y \prec pc] \mid y \neq x \wedge B^{\natural}(y) \neq \epsilon \wedge l_y \in B^{\natural}(y)\}) \\ B'^{\natural}(x) = l \cdot pc\end{array}} \ \text{(STORE)}$$

$$\frac{B^{\natural}(x) = l \cdot b}{B'^{\natural}(x) = b} \ \text{(FLUSH)}$$

$$\frac{stmt(pc) = fence}{\forall x. B'^{\natural}(x) = \epsilon} \ \text{(FENCE)}$$

$$\frac{stmt(pc) = cas\, x, r, s, q \qquad B^{\natural}(x) = l \qquad \psi = v}{\begin{array}{c}\psi' = v \vee \bigvee\{[l_y \prec pc] \mid y \neq x \wedge B^{\natural}(y) \neq \epsilon \wedge l_y \in B^{\natural}(y)\}) \\ B'^{\natural}(x) = \epsilon\end{array}} \ \text{(CAS)}$$

4.1 Avoiding Executions

When avoiding an execution, we would like to build all of the possible ways of repairing it. To accomplish that, we first define an ordering predicate that captures one way to repair an execution.

Ordering Predicates Given a pair of program labels l and k (in the same thread), we use $[l \prec k]$ to denote an ordering predicate. Informally, the intended meaning of $[l \prec k]$ is that in any execution involving these two labels, the statement at label l should have a visible effect before the statement at label k. Next we define what it means for an execution π to violate an ordering predicate $[l \prec k]$, denoted as $\pi \not\models [l \prec k]$.

An execution $\pi \not\models [l \prec k]$ if and only if: $\exists i, j.\ 0 \leq i < j < |\pi|$ such that:

1. $label(\pi_i) = l$, $label(\pi_j) = k$ and $tr(\pi_i) = tr(\pi_j)$.
2. $stmt(l) = store(r, x)$ and $stmt(k) \in \{store(p, y), load(y, p)\}$
3. $x \neq y$.
4. $\forall f.\ i < f < j$: $stmt(\pi_f) \neq flush(x)$, and $tr(\pi_f) = tr(\pi_i)$.

Informally, if we have an execution where a store is followed by some later store or a load (all by the same thread), where there is no flush of the first store to main memory in between the two operations, then the execution violates the predicate. The intuition is as follows: if the second operation is a read, then the read took effect before the first store became visible; if the second operation is a store, then, because flush can trigger at any point in an execution, it is possible to perform a flush of the second store immediately after the second store and make it visible before the first store is made visible. The above is a formulation for PSO. For TSO, we only need to change the second point to $stmt(k) \in \{load(y, p))\}$.

If π does not violate $[l \prec k]$, we say that π satisfies $[l \prec k]$, and write $\pi \models [l \prec k]$.

Computing a Repair for an Execution Given an execution, we would like to compute all possible ways to avoid it. We accomplish this by defining the instrumented semantics of Semantics 2 (here, we show the semantics for PSO, the semantics for TSO are similar). The semantics update an auxiliary map $B^{\natural}$ which records the program labels of statements accessing the buffer:

$$B^{\natural} \in Thread \rightarrow Shared \rightarrow Label^*$$

In the initial state, the buffers are empty, i.e. $\forall x. B^{\natural}(x) = \epsilon$, and $\psi = false$. Let us look at the STORE rule. The premise says that if the (auxiliary) buffer contains the sequence l, then (i) the program counter pc will be appended to the buffer, and (ii) we will create an ordering predicates between pc and each label belonging to other buffers with the same thread: the idea is that if there are pending stores in the other buffers (i.e. their labels are in their respective auxiliary buffer), then we can repair the current execution by ordering any of those stores before pc. The reason that we use $\vee$ when building ψ is that any of the enumerated ordering predicate is sufficient to repair the execution.

We note that even though in this work we only apply this repair procedure, by $avoid$ function in the algorithm, for a violating execution (under some specification), the procedure is independent of whether the execution is violating or not. Hence, we can easily use this procedure as-is for any execution. For instance, recent work has shown that it may be interesting to repair correct executions [31].

Computing a Repair for a Set of Executions Finally, in Algorithm 1, we combine repairs for each execution into a global repair formula φ. The reason we use $\wedge$ when combining individual avoid predicates is because we need to make sure that each execution is eliminated. When the algorithm terminates, the formula φ will contain all of the possible ways to repair all of the (violating) executions seen during this algorithm invocation. Next, we discuss how to enforce the formula φ into the program.

4.2 Enforcing Constraints

Given the repair formula φ, a satisfying assignment of the formula is a set of truth assignments to ordering predicates. Such an assignment represents a potential solution for all of the violating executions that contributed to the formula. There are various ways to enforce such an assignment in the program.

Enforce with Fences In this work, we chose to enforce the assignment $[l \prec k] : true$ by inserting a memory fence between the labels l and k. This prohibits the reordering of the memory accesses at these two labels. Algorithm 2 takes in a formula φ and a program, finds a minimal satisfying assignment to the formula and enforces the predicates with a fence. In practice, we insert a more specific fence (store-load or store-store) depending on whether the statement at k is a load or a store.

Enforce with Atomicity We can enforce $[l \prec k] : true$ as an atomic section that includes both labels l and k. Such an atomicity constraint (that may require more than two shared accesses) can be realized in various ways: with locks, or with hardware transaction memory (HTM) mechanisms [9] (HTM is especially suitable if the number of shared locations is bounded).

Enforce with CAS On TSO, we can enforce the fence with CAS to a dummy location. That is, we can use $cas(dummy, r, s, q)$, where $dummy$ is a location not used by the program (similarly r, s and q are never read in the thread). Regardless of whether such a CAS fails or succeeds on the dummy location, in order to proceed, it requires that the buffer is flushed (similarly to a fence). On PSO, it is also possible to use CAS to enforce the predicate, by having a CAS on the *same* location as the first store in the predicate, i.e., $cas(x, r, s, q)$. However, we would need to make sure that the CAS always fails (so does not modify x). Hence, enforcing order using CAS on PSO will only work when the contents of x and r are provably different.

5. Implementation

5.1 Architecture

Our framework is based on the LLVM infrastructure. To support our dynamic analysis and synthesis framework, we extended the LLVM interpreter (called *lli*) with the following features, not supported by the original LLVM framework:

Algorithm 2: $enforce(\varphi, P)$

```
Input: Formula $\varphi$, Program $P$
Output: Program $P'$ with fences that ensure an assignment
        satisfying $\varphi$
if $\varphi = true$ then return $P$
$I = \{[l_1 \prec k_1]...[l_n \prec k_n]\}$ be an assignment to predicates such
    that $I \models \varphi$ and $I$ is a minimal satisfying assignment of $\varphi$.
$P' = P$
foreach $[l \prec k] \in I$ do
    insert a $fence$ statement right after label $l$ in $P'$
return $P'$
```

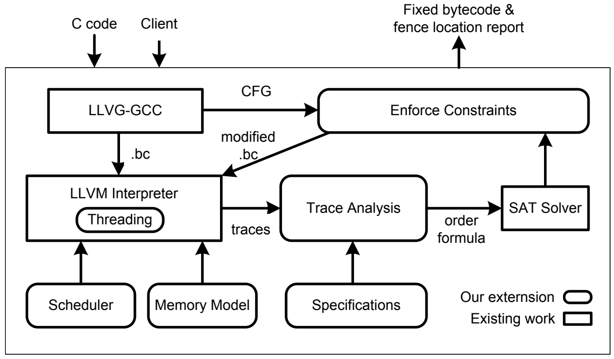


Figure 3: Schematic view of the structure of our tool. Rectangles denote existing work, while oval shapes denote our work.

- *Multi-Threading*: we added support for user-level threads. This includes support for statements *fork* and *join* as outlined in Section 3 as well as the *self* statement and per-thread local execution contexts.
- *Relaxed Memory Models*: we added support for both TSO and PSO memory models. This includes the relevant write buffers (outlined earlier), and extending LLVM intermediate instructions (e.g., stores and loads) to work with the memory model. We also support compare-and-swap (CAS) instructions, often used to implement locks and concurrent algorithms.
- *Scheduler*: we added scheduling support allowing us to plug-in different (demonic) schedulers for controlling the actions of the memory system (i.e., flushing) and thread operations.
- *Specifications*: we added support for checking memory safety errors such as array out of bounds and null dereferencing. We also added support for checking conditions such as linearizability and sequential consistency [14] (this required us to write a number of executable (sequential) specifications for all of the concurrent algorithms we analyzed).
- *Synthesis*: to repair a violating execution, we recorded all shared memory accesses in the violating execution. The resulting access history will be used to build a repair formula ψ as outlined in Semantics 2.

We believe that these extensions are of wide interest: for instance, we can use the framework for testing sequential programs for memory safety errors. Our extensions are parametric and the user simply selects the relevant parameters (e.g., which memory model, what properties to test) and runs their concurrent program with our version of the LLVM interpreter *lli*.

Figure 3 shows the overall structure of our synthesis framework. The input to DFENCE is a concurrent C algorithm and a (concurrent) client that calls the methods of the algorithm. LLVM-gcc then compiles both into a single bytecode file. Then, the interpreter can exercise the program with the given scheduler and memory model. The sequence of calls and returns that appear in the execution is collected, and is then checked against the given specification (e.g., linearizability). If the execution violates the specification, a repair for this single execution is calculated. A global repair for a set of violating executions can be calculated via a SAT solver and enforced into the given bytecode by inserting fences, producing a new bytecode representation. The new bytecode is fed back to the interpreter and the process repeats until no more violations are found or a user-specified timeout (or other bound e.g., number of times the program has run) is reached. All the phases of the flow in Figure 3 are completed by the framework automatically.

5.2 Extensions

The following provides additional details on each extension.

Multi-Threaded Support Supporting user level threads requires handling `fork`, `join`, and `self`. As outlined in Section 3, `fork` creates a new thread, `join` is used to wait for a thread to complete before proceeding and `self` returns the thread identifier of the caller. The reason we added `fork` and `join` is for handling clients that exercise concurrent algorithms. The reason we added `self` was because some concurrent algorithms require this method.

In the interpreter, we added a new map data structure, called *ThreadStacks* that maps a thread identifier to a list of execution contexts (stack frames). When the thread terminates, the vector becomes empty.

When `fork` is encountered, a new thread ID and a list of execution contexts is created in *ThreadStacks*. The execution of normal bytecodes proceed as-is (they access the particular execution context), and when a new function is called, a new execution context will be created. The call to `join` completes only if the target thread has finished, that is, its execution context list is empty, otherwise, the caller will block.

We added a function called *GetEnabledThreads* which returns the set of enabled threads. A thread is enabled if its list of execution contexts is not empty. This function is typically used by the scheduler to decide which thread to schedule next.

Relaxed Memory Model Support We maintain a write buffer for each thread (TSO) or each shared variable (PSO). The write buffer is simulated with a FIFO queue. The write buffers are managed by a dynamic map, and a new write buffer will be created when the memory address is *visited* for the first time. In this "lazy" approach, we do not need to change the functions that allocate memory such as `malloc` and `mmap`. Note that functions which deallocate memory, such as `free` and `munmap`, do not flush the write buffers.

Write buffers are only assigned to shared variables and thread-local variables access the memory directly. The implementation of shared variable accesses follows closely Semantics 1. To handle programs with locks, we implement lock acquire as a loop calling a CAS instruction to write 1 to the lock variable, and the function returns when CAS succeeds. Lock release writes 0 to the lock variable and returns. Both functions are wrapped with memory fences before and after the bodies, simulating the volatile feature of the lock variable.

Scheduler Support In our framework, the scheduler is designed as a plug-in, and is relatively independent of the interpreter. This allows experimenting with different schedulers for exposing violations.

To control scheduling, the probability of stores being delayed in the write buffer is gated by a parameter we call *flush probability*. The scheduler considers this parameter when deciding on the next step. At each step in the execution, based on the flush probability, the scheduler can decide which thread will take a step or which flushing action should be performed (for instance, for PSO, it can chose to flush only values for a particular variable):

- To reduce scheduling overhead, we use a form of partial-order reduction where a thread is not context switched if it keeps accessing local memory (thread-local variables).
- At each scheduling point, an enabled thread is selected randomly, and execution proceeds with the selected thread.
- Once a thread is selected, if its write buffers are empty, the thread makes a step. Otherwise, the scheduler randomly decides (with a user-provided flush probability) whether to flush the write buffer or proceed to the next step.

In our work, we found this strategy to be effective in exposing violations, but the framework allows the user to try out other scheduling strategies.

Specifications DFENCE supports both checking for memory safety properties such as array out of bounds, and null de-referencing, as well as properties such as linearizability and sequential consistency.

For space reasons, we do not elaborate on the precise definition of sequential consistency and linearizability. These, as well as extensive discussion comparing the two can be found in [14, Ch. 3.4-3.5]. Here we only discuss the essence of these conditions. Given a program execution, both of these criteria require us to first obtain a history (call it H) by extracting the sequence of calls and returns that appear in the execution. To check whether H is sequentially consistent, we compute two items: (i) sequentialization of H, call it $seq(H)$, and (ii) a witness history W that simply executes the relevant specification on the sequence of method calls appearing in $seq(H)$. If we find a sequentialization of H, where $W = seq(H)$, then we say that H is sequentially consistent, otherwise, if we cannot find any sequentialization of H which matches a witness, we say that H is not sequentially consistent. Linearizability is a more restricted variant of sequential consistency: it restricts how $seq(H)$ is computed. However, linearizability is compositional, while sequential consistency is not.

There are three key points to take away:

- Checking linearizability or sequential consistency requires a *semantic sequential specification* of the algorithm, e.g., specifying how the $push$ method of a stack behaves sequentially. Indeed, we had to provide such specifications for our experiments. The specifications are reusable for analyzing other concurrent algorithms: for instance, once we have the specification of a queue, we can use it to analyze any concurrent queue algorithm.
- Linearizability is a more restricted property than sequential consistency and is the one commonly used in the literature for concurrent algorithms. However, the interplay between sequential consistency, linearizability and relaxed memory models has not yet been well studied, and as we will see, interestingly, there are algorithms that look to be sequentially consistent, yet are non-lineraizable under relaxed memory models. The advantage of such algorithms is that they require less synchronization.
- Checking both of these properties requires computing all $seq(H)$'s of a history H, an operation clearly exponential in the length of H.

To check for linearizability and sequential consistency, the tool records a concurrent history and checks whether the concurrent history has a witness sequential history, as discussed above.

To detect memory safety violations, we keep auxiliary information such as the starting address and length, for global and heap memory units. For globals, this can be obtained by scanning the global segment of the intermediate bytecode. For heap, we can obtain the information from calls to `malloc` and `mmap` (the auxiliary state is deleted upon calling `free` or `munmap`).

When a load or a flush instruction occurs, the analysis checks whether the (to be) accessed address is within the current available memory (by checking in the auxiliary state). To reduce the overhead of the checking, we record each memory unit in a self balanced binary tree with the starting addresses as the keys. If we find that the target address is not inside any memory units, then a memory safety violation is triggered.

Synthesis Following Algorithm 1 from Section 4, once a violating execution π is discovered, the global avoid formula φ is updated to take avoiding π into account. In Algorithm 1, "?" indicates the non-deterministic choice of when the enforce procedure is invoked to produce a new program with a fix covering all violating executions accumulated so far. In DFENCE, we realized "?" as an iteration count, counting up to a user-defined threshold. To reduce overhead, DFENCE only records the relevant (shared memory accessing) instructions and not the whole execution.

To compute a satisfying assignment for the boolean formula φ, we use the off-the shelf MiniSAT [10] solver. Each (non-repeated) clause in the formula is assigned a unique integer. The satisfying assignment produced by MiniSAT has no guarantee of minimality. To obtain (all) minimal solutions, we call MiniSAT repeatedly to find out all solutions (when we find a solution, we adjust the formula to exclude that solution), and then we select the minimal ones.

Enforcing Once a minimal solution is selected, the labels in the ordering predicates are used to index the instruction pointers in the LLVM IR. The synchronization (e.g., fences) is then inserted into the IR by calling LLVM's instruction manipulation methods. The new IR is well-formed and is used for the next synthesis iteration. We optimize the insertion further by applying a merge phase which combines certain redundant fences: we apply a simple static analysis which eliminates a fence if it can prove that it always follows a previous fence statement in program order, with no store statements on shared variables occurring in between.

6. Evaluation

In this section, we evaluate the effectiveness of DFENCE in automatically inferring fences for realistic concurrent algorithms.

We used LLVM-GCC version 4.2 for compiling `C` programs into LLVM bytecode and LLVM runtime version 2.7 [19] for our implementation. Our implementation is designed to hook at the relevant places in LLVM and hence we expect it to be portable to newer versions. We used the open-source MiniSAT [10] solver.

6.1 Methodology

We applied DFENCE to a number of challenging concurrent `C` algorithms under different settings by varying four dimensions: (i) the memory model (TSO, PSO) (ii) the specification (memory safety, sequential consistency, and linearizability) (iii) the clients, and (iv) scheduler parameters.

In the first part of the evaluation, we evaluate the quality of fence inference under different choices along these four dimensions. Many of the algorithms used in our experiments have already been extensively tested on one memory model, usually TSO (x86). In this case, we first removed the fences from the algorithms and then ran DFENCE to see if it could infer them automatically. When we did not have the available fences for a given model, say PSO, we ran our tool and inspected the results manually to make sure the inferred fences were really required.

In the second part of our evaluation, we explore some of the connections between the different dimensions. In particular, we set out to answer the following questions:

1. **Memory Model:** How does the choice of memory model affect the choice of scheduler used to expose violations? How does it affect the choice of clients?
2. **Client:** How does the choice of client affect the ability to expose a violation, and how does it relate to the choice of specification and memory model?
3. **Specification:** How does the choice of specification affect the resulting fence placements?

6.2 Benchmarks

For our experiments, we used 13 concurrent C algorithms described in Tab. 2: five work-stealing queues (*WSQs*), three idempotent work-stealing queues (*iWSQ*'s), two queue algorithms, two set algorithms, and one lock-free concurrent memory allocator. All eight of the work-stealing queues share the same operations: `put`, `take` and `steal`. The Chase-Lev *WSQ* example has already been discussed in Section 2.

We chose *WSQ*s because they are a key technique for sharing work and are at the heart of many runtimes. We are not aware of any previous study which discusses extensive analysis or inference for *WSQ*s. Further, we believe that this is the first study to analyze and infer fences for Michael's lock-free memory allocator, arguably one of the largest and most complex concurrent C algorithms available. DFENCE consumes the C code directly and handles all low-level features of the implementation.

6.3 Results

Tab. 3 summarizes our results. Under each of the three specifications, we show the fences that DFENCE synthesized for a given memory model (we note that memory safety checking is always on, hence, Linearizability and Sequential Consistency columns include fences inferred due to memory safety violations). We describe each fence with the triple $(m, line1 : line2)$ to mean that we discovered the need for a fence in method m between lines $line1$ and $line2$. We use ”$-$” to indicate the end of the method in the triple.

For *Chase-Lev's WSQ*, we use the locations indicated in Fig. 1. For *Cilk's THE WSQ*, the line numbers correspond to our C implementation. For all *WSQ* and *iWSQ*, the line numbers are the same as those found in [24]. Similarly for the rest of the algorithms.

For all *iWSQ* algorithms, the "Memory Safety" column also includes the "no garbage tasks returned" specification. Analysis of *iWSQ* algorithms under Linearizability or SC requires more involved sequential specifications and is left as future work.

The last column shows the number of locations where a fence can be potentially inserted — this corresponds to the total number of store instructions in the LLVM bytecode. Note that this number is an order of magnitude greater than the actual number of inferred fences, meaning that without any help from the tool, we would need to manually examine an exponential number of location combinations. By Source Lines of Code (LOC), we mean the actual C code and by Bytecode LOC we mean the LLVM bytecode (which is a much larger number).

6.3.1 Additional and Redundant Fences

The *iWSQ* algorithms of [24] are designed to not require store-load fences in the owner's operations (on TSO). Our experimental results support this claim: the tool did not find the need for store-load fences on both TSO and PSO.

On PSO, for all three *iWSQ*'s, our tool finds the need for an additional "inter-operation" fence (store-store) at the end of `take`. In addition, for *FIFO iWSQ*, the tool suggests an additional "inter-operation“ fence at the end of `put`. We call these fences "inter-operation“ as they appear at the end of operations (and can even be placed right outside the end of a method return). These "inter-operation" fences are not mentioned in [24] and the C code was only tested on TSO. Also, for the C implementation of the *THE* algorithm, our tool discovered a redundant (store-load) fence in the `take` operation. We believe these results suggest that the tool may be helpful to a designer in porting concurrent algorithms to different memory models.

6.3.2 Number of Executions

In Algorithm 1, we used "?" to denote a non-deterministic choice of when to stop gathering violating executions and implement their repair. This lets us control the number of executions (say K) we gather before performing a repair on the incorrect subset of those executions. We refer to each repair phase as a round. Our analysis terminates whenever we see K executions and in those K executions there are no incorrect executions. For example, one scenario with $K = 1000$ per round can be:

- Round 1: run the program with 1000 executions, discover 500 bad executions and repair them using 2 fences.
- Round 2: run the new program again for 1000 executions, discover 100 bad execution and repair them using 1 fence.
- Round 3: run the new program again for 1000 executions and discover no bad executions. The analysis terminates (having introduced a total of three fences to repair the observed executions).

Fig. 4 shows how the number of executions per round affects the synthesis results of *Cilk's THE* algorithm with sequential consistency specification on the PSO memory model (the x-axis is log). The "multiple rounds" points should be interpreted as follows:

- Take a value on the x-axis, say K. This value stands for the number of executions per round.
- Take the corresponding value on the y-axis, say F. This value stands for the number of inferred fences. The number of rounds is always less than $F + 1$. The "+1" is to account for the last round which does not find any violations and verifies the fixing.
- For example, if we take $K = 1000$, the corresponding y-axis value is $F = 3$. Hence, the number of rounds is certainly ≤ 4.

We observe that with about 1000 executions per round, and with *at most* four rounds, we can infer all three of the required fences.

Alternatively, we can try to gather as many executions as possible and repair them all at once. In that experiment we set the number of rounds to one. The results are the "one round" points in the figure. These results indicate that we will need at least 200,000 executions to infer all of the three fences, an increase of about 65x in the number of executions one needs to consider!

The intuition is that once we find some violations and repair them, we eliminate many executions that need not be inspected further. If we do not eliminate violating executions relatively quickly, then we can keep seeing the same executions for a while (or executions caused by the same missing fence(s)), and it may take significanly longer to find all distinct violating executions. Therefore, DFENCE invokes the repair procedure after examining a (relatively) small number of executions.

6.4 Client

The choice of client is an important factor in our experiments. If the client cannot exercise the inputs which could trigger violations, even an ideal scheduler would be unable to expose errors.

Client vs. Specification The worst case time required for checking linearizability or sequential consistency of an execution is exponential in the length of the execution.

Algorithm	**Description**
Chase-Lev's WSQ [7]	`put`, `take` operate on the same end of the queue, `steal` works at the opposite end. Both `take` and `steal` implemented using CAS.
Cilk's THE WSQ [12]	`put` and `take` operate on the same end of the queue, `steal` works at the opposite end. Both `take` and `steal` use locks. This is the core algorithm used in the MIT's Cilk runtime.
LIFO iWSQ (Fig.1, [24])	`put`, `take` and `steal` operate on the same end of the queue. Only `steal` uses CAS.
FIFO iWSQ (Fig.2, [24])	`put` operates on one end of the queue, `take` and `steal` work at the opposite end. Only `steal` uses CAS.
Anchor iWSQ (Fig.3 [24])	`put` and `take` operate on the same end of the queue, `steal` works at the opposite end. Only `steal` uses CAS.
LIFO WSQ	same as *LIFO iWSQ* except that all operations use CAS.
FIFO WSQ	same as *FIFO iWSQ* except that `take` uses CAS to update the `head` variable.
Anchor WSQ	same as *Anchor iWSQ* except that all operations use CAS.
MS2 Queue [23]	`enqueue` and `dequeue` operate at different ends of a linked list. Both use locks.
MSN Queue [23]	`enqueue` and `dequeue` operate at different ends of a linked list. Both use CAS.
LazyList Set [13]	`add`, `contains` and `remove` operate on a sorted linked list. All three use locks.
Harris's Set [8]	`add`, `contains` and `remove` operate on a sorted linked list. All three are implemented with CAS.
Michael's Allocator [21]	The `C` lock-free memory allocator exports two functions: `malloc` and `free`. These operations use CAS.

Table 2: Algorithms used in our experiments. We use *iWSQ* as a shortcut for idempotent work stealing queues.

	Memory Safety		**Sequential Consistency**		**Linearizability**		**Source**	**Bytecode**	**Insertion**
Benchmark	TSO	PSO	TSO	PSO	TSO	PSO	**LOC**	**LOC**	**Points**
Chase-Lev's WSQ	0	0	F1	F1, F2	F1, F2	F1, F2, F3	150	696	96
Cilk's THE WSQ	0	0	(put, 11:13) (take, 5:7)	(put, 11:13) (take, 5:7) (steal, 6:8)	-	-	167	778	105
FIFO iWSQ	0	(put, 4:5) (put, 5:-) (take, 5:-)	-	-	-	-	149	686	102
LIFO iWSQ	0	(put, 3:4) (take, 4:-)	-	-	-	-	152	702	101
Anchor iWSQ	0	(put, 3:4) (take, 4:-)	-	-	-	-	162	843	107
FIFO WSQ	0	0	0	(put, 4:5) (put, 5:-)	(put, 4:5)	(put, 4:5) (put, 5:-)	143	789	91
LIFO WSQ	0	0	0	(put, 3:4)	0	(put, 3:4)	136	693	92
Anchor WSQ	0	0	0	(put, 3:4)	0	(put, 3:4)	152	863	101
MS2 Queue	0	0	0	0	0	0	62	351	46
MSN Queue	0	0	0	(enqueue, E3:E4)	0	(enqueue,E3:E4)	81	426	43
LazyList Set	0	0	0	0	0	0	121	613	68
Harris's Set	0	0	0	(insert, 8:9)	0	(insert, 8:9)	155	695	86
Michael's Memory Allocator	0	(MFNSB, 11:13) (DescAlloc, 5:8) (DescRetire, 2:4)	0	(MFNSB, 11:13) (DescAlloc, 5:8) (DescRetire, 2:4) (free, 16:18)	0	(MFNSB, 11:13) (DescAlloc, 5:8) (DescRetire, 2:4) (free, 16:18)	771	2699	244

Table 3: Fences inferred for each algorithm under different specifications and memory models. The table also shows the size of each algorithm in terms of Source LOC, LOC in bytecode, and insertion points. MFNSB stands for MallocFromNewSB. "0" in the table means that the tool did not find any fences; "-" means "cannot satisfy the property (like *Cilk's THE WSQ*)" or that "a specification is not yet available". For *WSQ*'s and *iWSQ*'s, the numbers do *not* include fences in the `expand` function.

For linearizability, the complexity can be reduced to linear if the programmer provides linearization points, as commonly required in verification [29]. However, for many algorithms finding linearization points is very difficult. Further, for properties such as sequential consistency, the concept of a linearization point is not applicable. This means that when we are checking these two properties, it is important to have the client produce relatively short executions, yet rich enough to expose violations.

Client vs. Coverage A good client should achieve good coverage, at the least, it would allow for all program points in each method to be visited. For example, in the case of a *WSQ*, it is important to explore schedules where concurrent threads interact in a queue that is sometimes empty and sometimes not. This is because different violations can occur in both cases. Thus it is important that a client can generate both cases.

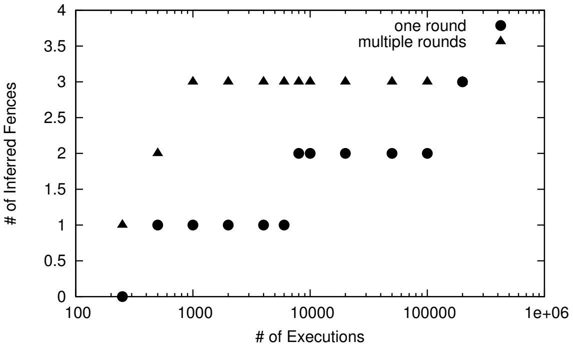


Figure 4: Number of Inferred Fences vs. Number of Rounds/Executions-per-Round.

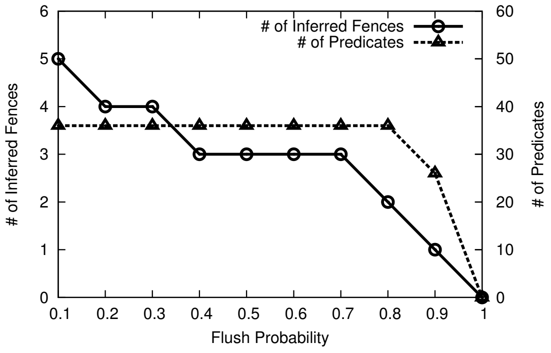


Figure 5: Effect of Flush Probability for Cilk's THE WSQ on PSO.

6.5 Scheduler vs. Memory Model

We find that setting the right flush probability (described in Sec. 5.2) is a key parameter for effectively triggering violations. With small values for the flush probability, meaning less flushing of the write buffers, we expose more violations and hence synthesize more missing fences, than we do with greater values. If the flush probability is too high, then the behavior of the program becomes as if the memory model is sequentially consistent (the intuition is that we are flushing more frequently, so the buffers are mostly empty), and hence it is more difficult to find violations.

Fig. 5 illustrates how the number of synthesized fences correlates with the flush probability for *Cilk's THE WSQ* under PSO with $K = 1000$. We can observe two tendencies:

- When the flush probability is large, i.e., greater than 0.8, the number of discovered ordering predicates decreases. With fewer predicates, some required fences cannot be inferred.
- When the flush probability is small, i.e., less than 0.4, the same unnecessary predicates may appear in most buggy executions, causing redundant fences. Thus the number of redundant fences is increased.

There are two ways to eliminate redundant fences. First, we can increase the flush probability. Second, we can collect more violating executions by increasing the execution number per round. If one of the violating executions does not contain the reordering which caused the generation of a redundant fence, then the redundant fence will not be inferred.

Flush Probability vs. PSO For PSO, we found that a flush probability around 0.5 is suitable for optimal inference for all benchmarks and specifications. With this value, the number of executions should be set high enough to collect sufficient predicates. For an accurate inference, we find that the number of collected violating executions should be around 10 times greater than the maximum number of possible predicates collected for the program. This maximum number can be obtained by testing executions with a low flush probability, and in Fig. 5 this number is 36.

Flush Probability vs. TSO Interestingly, we find that the optimal flush probability for the same program on TSO can be quite a bit smaller than the value used for PSO. In our experiments, the flush probability for TSO is usually set to 0.1. The intuition is that for the same program, the number of write buffers in TSO is usually much smaller than the number of buffers in PSO. This means that with the same flush probability, the chance that a buffer is flushed in TSO is higher than in PSO. It follows that if we use too high of a flush probability for write buffers, we are less likely to find violations caused by memory model reorderings.

We find that for the same specification, under PSO, the program requires more fences than under TSO. This is reasonable, since PSO allows both store-store and store-load reordering, while TSO only allows a store-load reordering.

6.6 Fences vs. Specification

Memory Safety is Ineffective As we see in the table of results, and perhaps counter-intuitively, we find that memory safety specifications are almost always not sufficiently strong to trigger violations. For instance, with *WSQ*'s, this is because a violating execution is more often exhibited in the form of losing an item or in returning duplicate items, rather than in accessing out of bounds memory. We note that one trick that may make memory safety more effective in triggering violations is to use a specific client: instead of elements of a primitive type, one stores pointers to newly allocated memory in the queue. Then, the client frees the pointer immediately after it has fetched it from the queue. In that way, one may be able to detect duplicate items. We leave this experiment as future work.

We find that safety specifications are useful for the memory allocator as there are many pointer dereferences performed in the code, and the buffering of an updated pointer value can cause a dereference of `NULL` pointer in another thread.

Linearizability vs. Sequential Consistency Another interesting point is that for the same memory model, linearizability generally requires more fences than sequential consistency. On one side, this can be expected as linearizability is a stronger property, but on the other, it is not obvious that by slightly weakening linearizability to sequential consistency, one does not end up with a completely "incorrect" algorithm (in the sense of losing items for *WSQ*'s or violating memory safety). For the four *WSQ*'s (excluding *THE*), the fence added to guarantee linearizability is used to trigger the flushing at the linearization point.

Algorithm without fences on TSO Interestingly, by weakening the correctness criteria from linearizability to sequential consistency, we obtain an algorithm, *FIFO WSQ* on TSO, for which the tool does not find any fences (except the slow path `expand` function). Note that unlike *iWSQ* algorithms, we did not weaken the actual specification of *FIFO*, that is, we did not need an idempotent variant of *FIFO*. Our finding is also inline with a recent result which shows that it is impossible to eliminate certain orders with linearizability [3], but by slightly weakening it, that may be possible.

Cilk's THE *is not linearizable* As our tool can check properties without any relaxed memory model effects, we were able to detect that *Cilk's THE WSQ* is not linearizable with a deterministic sequential specification (even without any relaxed memory model effects). However, the tool indicates that the algorithm is sequentially consistent and is indeed accepted to be correct [1]. DFENCE infers the necessary fences under the sequential consistency specification.

6.7 Memory Allocator

For the memory allocator, we used the client sequence *mmmfff* | *mfmf*, where *m* stands for malloc and *f* stands for free. In this sequence, the free function always frees up the oldest memory unit allocated by the same thread. As far as we know, no previous work can synthesize synchronization for concurrent algorithms of this scale. For the allocator, we found that three fences were required to satisfy the memory safety property, i.e., running to termination without segmentation fault. If we specified the additional criteria such as sequential consistency or linearizability, one additional fence in the `free` function is inferred. All four of the fences that our tool inferred are discussed in the paper which describes the memory allocator algorithm [21]. The `C` implementation of the allocator has 11 store-store memory fences and DFENCE was able to infer only 4 of them. The problem of how to uncover the other 7 fences is left as future work.

7. Related Work

Exhaustive Approaches for Relaxed Memory Models The works in [16, 25] describe explicit-state model checking for the Sparc RMO model, but neither of these approaches perform fence inference. The work in [15] describes an explicit-state model checker and a simple inference technique for the .NET memory model. Recent work [17, 18] focuses on fence inference based on model checking (with abstraction). In [4], a different approach is taken where instead of working with explicit state, they convert programs into a form that can be checked against an axiomatic model specification. Collectively, the fundamental limitation of these exhaustive approaches is that they are not inherently scalable for larger concurrent programs and often require manual intervention (to specify abstractions or a model of the program). In contrast, our focus is squarely on handling large concurrent C implementations, which in turn dictates our dynamic synthesis approach.

Delay Set Analysis A number of works on fence inference [11, 20, 27] rely on concepts such as delay sets and conflict graphs [26]. Particularly, the Pensieve project implements fence inference based on delay set analysis. This kind of analysis is necessarily more conservative than ours for two reasons: first, it is static in nature, and second, the correctness criteria are more restrictive than the ones used in this work. For instance, their criteria would prevent behaviors that are linearizable and should be allowed.

Other Approaches The works in [5, 6] present algorithms that find violations under the TSO and PSO memory models based on correctness criteria similar to the ones used in the delay set analysis approaches mentioned above, meaning that [5, 6] can be needlessly conservative. Further, both of these works do not support fence inference. In [30], the authors propose algorithms that automatically infer synchronization constructs such as atomic sections. Their work does not deal with relaxed memory models.

8. Conclusion

We introduced dynamic synthesis for concurrent programs running on relaxed memory models. The key idea is to break the scalability barrier of standard synthesis approaches by instead focusing on dynamic executions: when a set of violating executions is discovered, the program is automatically repaired to avoid these executions, and the synthesis procedure continues with the repaired program.

We implemented our techniques in LLVM and evaluated its effectiveness on a comprehensive list of concurrent C algorithms including a complex lock-free memory allocator. Each algorithm was evaluated with two memory models (TSO and PSO), three specifications (linearizability, sequential consistency and memory safety), various repair strategies, and different scheduling parameters (e.g. flush probability). We demonstrated the effectiveness of our approach by automatically synthesizing necessary fences for each of the concurrent algorithms.

As future work, we plan to extend our tool with more advanced demonic schedulers to discover violations even quicker. We also plan to add support for other memory models and to evaluate our tool on a wider set of concurrent C programs.

References

[1] Personal Communication with the Cilk Team, 2011.

[2] ADVE, S. V., AND GHARACHORLOO, K. Shared memory consistency models: A tutorial. *IEEE Computer 29* (1995), 66–76.

[3] ATTIYA, H., GUERRAOUI, R., HENDLER, D., KUZNETSOV, P., MICHAEL, M. M., AND VECHEV, M. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL'11* (New York, NY, USA), ACM, pp. 487–498.

[4] 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.

[5] BURCKHARDT, S., AND MUSUVATHI, M. Effective program verification for relaxed memory models. In *CAV* (2008), pp. 107–120.

[6] BURNIM, J., SEN, K., AND STERGIOU, C. Testing concurrent programs on relaxed memory models. In *ISSTA* (2011), pp. 122–132.

[7] CHASE, D., AND LEV, Y. Dynamic circular work-stealing deque. In *SPAA* (2005), pp. 21–28.

[8] DETLEFS, D. L., FLOOD, C. H., GARTHWAITE, A. T., GARTHWAITE, E. T., MARTIN, P. A., SHAVIT, N. N., JR., AND STEELE, G. L. Even better dcas-based concurrent deques. In *DISC'00*.

[9] DICE, D., LEV, Y., MARATHE, V. J., MOIR, M., NUSSBAUM, D., AND OLESZEWSKI, M. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *SPAA'10*, ACM, pp. 325–334.

[10] EÉN, N., AND SÖRENSSON, N. An extensible sat-solver. In *SAT* (2003), pp. 502–518.

[11] FANG, X., LEE, J., AND MIDKIFF, S. P. Automatic fence insertion for shared memory multiprocessing. In *ICS* (2003), pp. 285–294.

[12] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In *PLDI '98*.

[13] HELLER, S., HERLIHY, M., LUCHANGCO, V., MOIR, M., SCHERER, W., AND SHAVIT, N. A lazy concurrent list-based set algorithm. In *OPODIS '05*, pp. 3–16.

[14] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Apr. 2008.

[15] HUYNH, T. Q., AND ROYCHOUDHURY, A. Memory model sensitive bytecode verification. *Form. Methods Syst. Des. 31*, 3 (2007).

[16] JONSSON, B. State-space exploration for concurrent algorithms under weak memory orderings. *SIGARCH Comput. Archit. News 36*, 5 (2008), 65–71.

[17] KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Automatic inference of memory fences. In *FMCAD'10*.

[18] KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Partial-coherence abstractions for relaxed memory models. In *PLDI'11*, pp. 187–198.

[19] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*, pp. 75–87.

[20] LEE, J., AND PADUA, D. A. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput. 50*, 8 (2001), 824–833.

[21] MICHAEL, M. M. Scalable lock-free dynamic memory allocation. In *PLDI '04* (2004), pp. 35–46.

[22] MICHAEL, M. M., AND SCOTT, M. L. Correction of a memory management method for lock-free data structures. Tech. rep., 1995.

[23] MICHAEL, M. M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC* (1996), pp. 267–275.

[24] MICHAEL, M. M., VECHEV, M. T., AND SARASWAT, V. A. Idempotent work stealing. In *PPoPP* (2009), pp. 45–54.

[25] PARK, S., AND DILL, D. L. An executable specification and verifier for relaxed memory order. *IEEE Trans. on Computers 48* (1999).

[26] 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.

[27] SURA, Z., WONG, C., FANG, X., LEE, J., MIDKIFF, S., AND PADUA, D. Automatic implementation of programming language consistency models. *LNCS 2481* (2005), 172.

[28] SUTTER, H., AND LARUS, J. Software and the concurrency revolution. *Queue 3*, 7 (2005), 54–62.

[29] VECHEV, M., YAHAV, E., AND YORSH, G. Experience with model checking linearizability. In *SPIN'09*, pp. 261–278.

[30] VECHEV, M., YAHAV, E., AND YORSH, G. Abstraction-guided synthesis of synchronization. In *POPL* (2010).

[31] WEERATUNGE, D., ZHANG, X., AND JAGANATHAN, S. Accentuating the positive: atomicity inference and enforcement using correct executions. In *OOPSLA'11*, pp. 19–34.