Fence Synthesis under the C11 Memory Model

Sanjana Singh[1], Divyanjali Sharma[1] Ishita Jaju[2], and Subodh Sharma[1]

[1] Indian Institute of Technology Delhi, India,
{sanjana.singh,divyanjali,svs}@cse.iitd.ac.in,
[2] Uppsala University, Sweden

Abstract. The C/C++11 (*C11*) standard offers a spectrum of ordering guarantees on memory access operations. The combinations of such orderings pose a challenge in developing *correct* and *efficient* weak memory programs. A common solution to preclude those program outcomes that violate the correctness specification is using *C11* synchronization-fences, which establish ordering on program events. The challenge is in choosing a combination of fences that (i) restores the correctness of the input program, with (ii) as little impact on efficiency as possible (*i.e.*, the smallest set of weakest fences). This problem is the *optimal fence synthesis* problem and is NP-hard for straight-line programs. In this work, we propose the first fence synthesis technique for *C11* programs called `FenSying` and show its optimality. We additionally propose a near-optimal efficient alternative called `fFenSying`. We prove the optimality of `FenSying` and the soundness of `fFenSying` and present an implementation of both techniques. Finally, we contrast the performance of the two techniques and empirically demonstrate `fFenSying`'s effectiveness.

Keywords: *C11*, fence-synthesis, optimal

1 Introduction

Developing weak memory programs requires careful placement of fences and memory barriers to preserve ordering between program instructions and exclude undesirable program outcomes. However, computing the correct combination of the type and location of fences is challenging. Too few or incorrectly placed fences may not preserve the necessary ordering, while too many fences can negatively impact the performance. Striking a balance between preserving the correctness and obtaining performance is highly non-trivial even for expert programmers.

This paper presents an automated fence synthesis solution for weak memory programs developed using the C/C++11 standard (*C11*). *C11* provides a spectrum of ordering guarantees called *memory orders*. In a program, a memory access operation is associated with a memory order which specifies how other memory accesses are ordered with respect to the operation. The memory orders range from *relaxed* (`rlx`) (that imposes no ordering restriction) to *sequentially-consistent* (`sc`) (that may restore sequential consistency). Understanding all the subtle complexities of *C11* orderings and predicting the program outcomes can quickly become exacting. Consider the program `(RWRW)` (§2), where the orders are shown as subscripts. When all the memory accesses are ordered `rlx`, there exists a program outcome that violates the correctness specification (specified as an *assert* statement). However, when all accesses are ordered `sc`, the program is provably correct.

In addition, the *C11* memory model supports *C11 fences* that serve as tools for imposing ordering restrictions. Notably, *C11* associates fences with memory orders, thus, supporting various degrees of ordering guarantees through fences.

This work proposes an *optimal* fence synthesis technique for *C11* called `FenSying`. It involves finding solutions to two problems: (i) computing an optimal (minimal) set of locations to synthesize fences and (ii) computing an optimal (weakest) memory order to be associated with the fences (formally defined in §3). `FenSying` takes as input *all* program runs that violate user-specified assertions and attempts optimal *C11* fence synthesis to stop the violating outcomes. `FenSying` reports when *C11* fences alone cannot fix a violation. In general, computing a minimal number of fences with multiple types of fences is shown to be NP-hard for straight-line programs [24]. We note, rather unsurprisingly, that this hardness manifests in the proposed optimal fence synthesis solution even for the simplest *C11* programs. Our experiments (§7) show an exponential increase in the analysis time with the increase in the program size.

Further, to address scalability, this paper proposes a *near-optimal* fence synthesis technique called `fFenSying` (`f`ast `FenSying`) that fixes *one* violating outcome at a time optimally. Note that fixing one outcome optimally may not guarantee optimality across all violating outcomes. In the process, this technique may add a small number of extra fences than what an optimal solution would compute. Our experiments reveal that `fFenSying` performs exponentially better than `FenSying` in terms of the analysis time while adding no extra fences in over 99.5% of the experiments.

Both `FenSying` and `fFenSying`, compute the solution from a set of combinations of fences that can stop the violating outcomes, also called *candidate solutions*. The candidate solutions are encoded in a *head-cycle-free* `CNF` `SAT` query [8]. Computing an optimal solution from candidates then becomes finding a solution to a *min-model finding problem*.

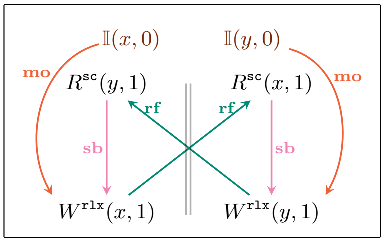
Many prior works have focused on automating fence synthesis (discussed in §8). However, the techniques presented in this paper are distinct from prior works in the following two ways: (i) prior techniques do not support C11 memory orders, and (ii) the proposed techniques in this paper synthesize fences that are portable and not architecture-specific.

Contributions. To summarize, this work makes the following contributions:

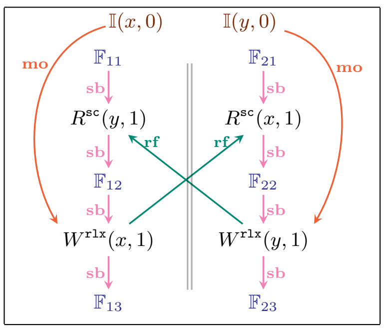
- The paper presents `FenSying` and `fFenSying` (§6). To the best of our knowledge, these are the first fence synthesis techniques for *C11*.
- The paper shows (using Theorems 1 and 2) that the techniques are sound, *i.e.*, if the input program can be fixed by *C11* fences, then the techniques will indeed find a solution. The paper also shows (using Theorem 3) that `FenSying` produces an optimal result in the number and type of fences.

Initially: $x = 0, y = 0$		
$a :=_{\texttt{sc}} y$	$\parallel$	$b :=_{\texttt{sc}} x$
$x :=_{\texttt{rlx}} 1$	$\parallel$	$y :=_{\texttt{rlx}} 1$
assert$(\neg(a{=}1 \land b{=}1))$		

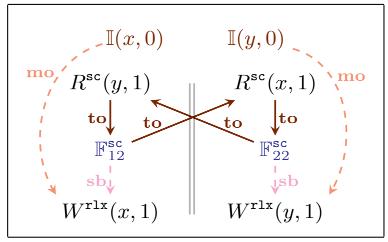
`RWRW`



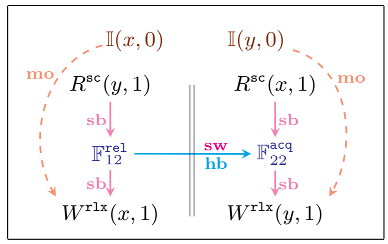
`RWRW-bt`



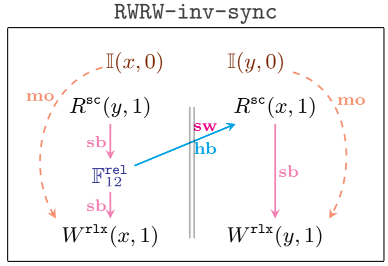
`RWRW-imm`



`RWRW-inv-to`



`RWRW-inv-sync`



`RWRW-inv-sync-opt`

– Finally, the paper presents an implementation of the said techniques and presents an empirical validation using a set of 1389 litmus tests. Further, the paper empirically shows the effectiveness of `fFenSying` on a set of challenging benchmarks from prior works. `fFenSying` performs on an average 67x faster than `FenSying`.

2 Overview of `FenSying` and `fFenSying`

Given a program P, a *trace* τ of P (formally defined in §3); is considered *buggy* if it violates an assertion of P. `FenSying` takes *all* buggy traces of P as input. The difference in `fFenSying` is that the input is a *single* buggy trace of P.

Consider the input program `(RWRW)`, where x and y are shared objects with initial values 0, and a and b are local objects. Let $W^m(o, v)$ and $R^m(o, v)$ represent the write and read of object o and value v with the memory order m. Let $\mathbb{I}(o, v)$ represent the initialization event for object o with value v. The parallel bars ($\|$) represent the parallel composition of events from separate threads. Figure `(RWRW-bt)` represents a buggy trace τ of `(RWRW)` under *C11* semantics. For convenience, the relations – *sequenced-before* ($\rightarrow^{\text{sb}}_{\tau}$), *reads-from* ($\rightarrow^{\text{rf}}_{\tau}$), *modification-order* ($\rightarrow^{\text{mo}}_{\tau}$) (formally defined in §3,§4) – among the events of τ are shown. The assert condition of (RWRW) is violated as the read events are not ordered before the write events of the same object, allowing reads from *later* writes.

Consider the following three sets of fences that can invalidate the trace (RWRW-bt): $c_1 = \{\mathbb{F}^{\mathtt{sc}}_{12}, \mathbb{F}^{\mathtt{sc}}_{22}\}$, $c_2 = \{\mathbb{F}^{\mathtt{rel}}_{12}, \mathbb{F}^{\mathtt{acq}}_{22}\}$ and $c_3 = \{\mathbb{F}^{\mathtt{rel}}_{12}\}$ (the superscripts indicate the memory orders and the subscripts represent the synthesis locations for fences). The solutions are depicted in Figures (RWRW-inv-to), (RWRW-inv-sync) and (RWRW-inv-sync-opt) for c_1, c_2 and c_3, respectively. The candidate solution c_1 prevents a total order on the `sc` ordered events [9,12], thus, invalidating (RWRW-inv-to) under *C11* semantics. With candidate solution c_2, a happens-before ($\xrightarrow[\tau]{\mathbf{hb}}$) ordering is formed (refer to §4) between $R^{\mathtt{sc}}(y, 1)$ and $W^{\mathtt{sc}}(y, 1)$. This forbids a read from an ordered-later write, thus, invalidating (RWRW-inv-sync). Candidate solution c_3 establishes a similar $\xrightarrow[\tau]{\mathbf{hb}}$ ordering by exploiting the strong memory order of $R^{\mathtt{sc}}(x, 1)$ and invalidates (RWRW-inv-sync-opt).

The candidate solution c_2 is preferred over c_1 as it contains weaker fences. On the other hand, candidate c_3 represents an optimal solution as it uses the smallest number of weakest fences. We formally define the optimality of fence synthesis in §3. While FenSying will compute the solution c_3, fFenSying may compute one from the many candidate solutions.

Both FenSying and fFenSying start by transforming each buggy trace τ to an *intermediate* version, $\tau^{\mathtt{imm}}$, by inserting *untyped C11* fences (called *candidate fences*) above and below the trace events. (RWRW-imm) shows an intermediate version corresponding to (RWRW-bt). The addition of fences (assuming they are of the strongest variety) leads to the creation of new $\xrightarrow[\tau]{\mathbf{hb}}$ ordering edges. This may result in cycles in the dependency graph under the *C11* semantics (explained in §4). The set of fences in a cycle constitutes a *candidate solution*. For example, an ordering from $\mathbb{F}_{12}$ to $\mathbb{F}_{22}$ in (RWRW-imm) induces a cyclic relation $W^{\mathtt{rlx}}(y, 1)$ $\xrightarrow[\tau]{\mathbf{rf}} R^{\mathtt{sc}}(y, 1) \xrightarrow[\tau]{\mathbf{hb}} W^{\mathtt{rlx}}(y, 1)$ violating the $\xrightarrow[\tau]{\mathbf{rf}}; \xrightarrow[\tau]{\mathbf{hb}}$ irreflexivity (refer to §4).

The candidate solutions are collected in a `SAT` query (Φ). Assuming c_1, c_2 and c_3 are the only candidate solutions for (RWRW-bt), then $\Phi = (\mathbb{F}_{12} \wedge \mathbb{F}_{22}) \vee (\mathbb{F}_{12} \wedge \mathbb{F}_{22}) \vee (\mathbb{F}_{12})$, where for a fence $\mathbb{F}^{m}_{i}$, $\mathbb{F}_i$ represents the same fence with unassigned memory order. fFenSying uses a `SAT` solver to compute the *min-model* of Φ, $\mathtt{min}\Phi = \{\mathbb{F}_{12}\}$. Further, fFenSying applies the *C11* ordering rules on fences to determine the weakest memory order for the fences in $\mathtt{min}\Phi$. For instance, $\mathbb{F}_{12}$ in $\mathtt{min}\Phi$ is computed to have the order `rel` (explained in §6). fFenSying then inserts $\mathbb{F}_{12}$ with memory order `rel` in (RWRW) at the location depicted in (RWRW-inv-sync-opt). This process repeats for the next buggy trace.

In contrast, since FenSying works with all buggy traces at once, it requires the conjunction of the `SAT` queries Φ_i corresponding to each buggy trace τ_i. The min-model of the conjunction is computed, which provides optimality.

3 Preliminaries

Consider a multi-threaded *C11* program (P). Each thread of P performs a sequence of *events* that are runtime instances of memory access operations (reads, writes, and rmws) on shared objects and *C11* fences. Note that an event is uniquely identified in a trace; however, multiple events may be associated with the same program location. The events may be atomic or non-atomic.

***C11* memory orders.** The atomic events and fence operations are associated with memory orders that define the ordering restriction on atomic and non-atomic events around them. Let $\mathcal{M} = \{\texttt{na}, \texttt{rlx}, \texttt{rel}, \texttt{acq}, \texttt{ar}, \texttt{sc}\}$, represent the orders relaxed (`rlx`), release (`rel`), acquire/consume (`acq`), acquire-release (`ar`) and sequentially consistent (`sc`) for atomic events. A non-atomic event is recognized by the `na` memory order. Let $\sqsubset \subseteq \mathcal{M} \times \mathcal{M}$ represent the relation *weaker* such that $m_1 \sqsubset m_2$ represents that the m_1 is weaker than m_2. As a consequence, annotating an event with m_2 may order two events that remain unordered with m_1. The orders in $\mathcal{M}$ are related as $\texttt{na} \sqsubset \texttt{rlx} \sqsubset \{\texttt{rel}, \texttt{acq}\} \sqsubset \texttt{ar} \sqsubset \texttt{sc}$. We also define the relation $\sqsubseteq$ to represent *weaker or equally weak*. Similarly, we define $\sqsupset$ to represent *stronger* and $\sqsupseteq$ to represent *stronger or equally strong*.

We use $\mathcal{E}^{\mathbb{W}} \subseteq \mathcal{E}$ to denote the set of events that perform write to shared memory objects *i.e.*, write events or rmw events. Similarly, we use $\mathcal{E}^{\mathbb{R}} \subseteq \mathcal{E}$ to denote events that read from a shared memory object *i.e.*, read events and rmw events, and $\mathcal{E}^{\mathbb{F}}$ to denote the fence events. We also use $\mathcal{E}^{(m)} \in \mathcal{E}$ (and accordingly $\mathcal{E}^{\mathbb{W}(m)}$, $\mathcal{E}^{\mathbb{R}(m)}$ and $\mathcal{E}^{\mathbb{F}(m)}$) to represent the events with the memory order $m \in \mathcal{M}$; as an example $\mathcal{E}^{\mathbb{F}(\texttt{sc})}$ represents the set of fences with the memory order `sc`.

Definition 1 (Trace). A *trace*, τ, of P is a tuple $\langle \mathcal{E}_\tau, \xrightarrow{\textbf{hb}}_\tau, \xrightarrow{\textbf{mo}}_\tau, \xrightarrow{\textbf{rf}}_\tau \rangle$, where

$\mathcal{E}_\tau$ $\subseteq \mathcal{E}$ represents the set of events in the trace τ;

$\xrightarrow{\textbf{hb}}_\tau$ (*Happens-before*) $\subseteq \mathcal{E}_\tau \times \mathcal{E}_\tau$ is a partial order which captures the event interactions and inter-thread synchronizations, discussed in §4;

$\xrightarrow{\textbf{mo}}_\tau$ (*Modification-order*) $\subseteq \mathcal{E}_\tau^{\mathbb{W}} \times \mathcal{E}_\tau^{\mathbb{W}}$ is a total order on the writes of an object;

$\xrightarrow{\textbf{rf}}_\tau$ (*Reads-from*) $\subseteq \mathcal{E}_\tau^{\mathbb{W}} \times \mathcal{E}_\tau^{\mathbb{R}}$ is a relation from a write event to a read event signifying that the read event takes its value from the write event in τ.

Note that, we use $\mathcal{E}_\tau^{\mathbb{W}}$, $\mathcal{E}_\tau^{\mathbb{R}}$ and $\mathcal{E}_\tau^{\mathbb{F}}$ (and also $\mathcal{E}_\tau^{\mathbb{W}(m)}$, $\mathcal{E}_\tau^{\mathbb{R}(m)}$ and $\mathcal{E}_\tau^{\mathbb{F}(m)}$ where $m \in \mathcal{M}$) for the respective sets of events for a trace τ.

Relational Operators. R^{-1} represents the inverse and R^+ represents the transitive closure of a relation R. Further, $R_1; R_2$ represents the composition of relations R_1 and R_2. Let $R|_{\texttt{sc}}$ represent a subset of a relation R on `sc` ordered events; *i.e.* $(e_1, e_2) \in R|_{\texttt{sc}} \iff (e_1, e_2) \in R \wedge e_1, e_2 \in \mathcal{E}^{(\texttt{sc})}$. Note that we also use the infix notation $e_1 R e_2$ for $(e_1, e_2) \in R$. Lastly, a relation R has a cycle (or is cyclic) if $\exists e_1, e_2 \in \mathcal{E}$ s.t. $e_1 R e_2 \wedge e_2 R e_1$.

A note on optimality. The notion of optimality may vary with context. Consider two candidate solutions $\{\mathbb{F}_i^{\texttt{sc}}\}$ and $\{\mathbb{F}_j^{\texttt{rel}}, \mathbb{F}_k^{\texttt{acq}}\}$ where the superscripts represent the memory orders. The two solutions are incomparable under *C11*, and their performance efficiency is subject to the input program and the underlying architecture. `FenSying` chooses a candidate solution c as an optimal solution if: (i) c has the smallest number of candidate fences, and (ii) each fence of c has the weakest memory order compared to other candidate solutions that satisfy (i).

Let $sz(c)$ represent the size of the candidate solution c and given the set of all candidate solutions $\{c_1, ..., c_n\}$ to fix P, let $\underline{sz}(P) = \texttt{min}(sz(c_1), ..., sz(c_n))$. Further, we assign weights $wt(c)$ to each candidate solution c, computed as the summation of the weights of its fences where a fence ordered `rel` or `acq` is assigned the weight 1, a fence ordered `ar` is assigned 2, and a fence ordered `sc` is assigned 3. Optimality for `FenSying` is formally defined as:

Definition 2. Optimality of fence synthesis. Consider a set of candidate solutions $c_1, ..., c_n$. A solution c_i (for $i \in [1, n]$) is considered optimal if:
(i) $sz(c_i) = \underline{sz}(P) \wedge$ (ii) $\forall j \in [1, n]$ s.t. $sz(c_j) = \underline{sz}(P)$, $wt(c_i) \leq wt(c_j)$.

4 Background: C11 Memory Model

The *C11* memory model defines a trace using a set of event relations, described in Definition 1. The most significant relation that defines a *C11* trace τ is the irreflexive and acyclic happens-before relation, $\rightarrow_\tau^{\mathbf{hb}} \subseteq \mathcal{E}_\tau \times \mathcal{E}_\tau$. The $\rightarrow_\tau^{\mathbf{hb}}$ relation is composed of the following relations [12].

$\rightarrow_\tau^{\mathbf{sb}}$ (*Sequenced-before*): total occurrence order on the events of a thread.

$\rightarrow_\tau^{\mathbf{sw}}$ (*Synchronizes-with*) Inter-thread synchronization between a write e_w (ordered $\sqsupseteq$ `rel`) and a read e_r (ordered $\sqsupseteq$ `acq`) when $e_w \rightarrow_\tau^{\mathbf{rf}} e_r$.

$\rightarrow_\tau^{\mathbf{dob}}$ (*Dependency-ordered-before*): Inter-thread synchronization between a write e_w (ordered $\sqsupseteq$ `rel`) and a read e_r (ordered $\sqsupseteq$ `acq`) when $e'_w \rightarrow_\tau^{\mathbf{rf}} e_r$ for $e'_w \in$ *release-sequence*[3] of e_w in τ [9,12].

$\rightarrow_\tau^{\mathbf{ithb}}$ (*Inter-thread-hb*): Inter-thread relation computed by extending $\rightarrow_\tau^{\mathbf{sw}}$ and $\rightarrow_\tau^{\mathbf{dob}}$ with $\rightarrow_\tau^{\mathbf{sb}}$.

$\rightarrow_\tau^{\mathbf{hb}}$ (*Happens-before*): Inter-thread relation defined as $\rightarrow_\tau^{\mathbf{sb}} \cup \rightarrow_\tau^{\mathbf{ithb}}$.

The $\rightarrow_\tau^{\mathbf{hb}}$ relation along with the $\rightarrow_\tau^{\mathbf{mo}}$ and $\rightarrow_\tau^{\mathbf{rf}}$ relations (Definition 1) is used in specifying the set of six `coherence conditions` [12,17]:

$\rightarrow_\tau^{\mathbf{hb}}$ is irreflexive. (`co-h`)

$\rightarrow_\tau^{\mathbf{rf}}; \rightarrow_\tau^{\mathbf{hb}}$ is irreflexive. (`co-rh`)

$\rightarrow_\tau^{\mathbf{mo}}; \rightarrow_\tau^{\mathbf{hb}}$ is irreflexive. (`co-mh`)

$\rightarrow_\tau^{\mathbf{mo}}; \rightarrow_\tau^{\mathbf{rf}}; \rightarrow_\tau^{\mathbf{hb}}$ is irreflexive. (`co-mrh`)

$\rightarrow_\tau^{\mathbf{mo}}; \rightarrow_\tau^{\mathbf{hb}}; \rightarrow_\tau^{\mathbf{rf}^{-1}}$ is irreflexive. (`co-mhi`)

$\rightarrow_\tau^{\mathbf{mo}}; \rightarrow_\tau^{\mathbf{rf}}; \rightarrow_\tau^{\mathbf{hb}}; \rightarrow_\tau^{\mathbf{rf}^{-1}}$ is irreflexive. (`co-mrhi`)

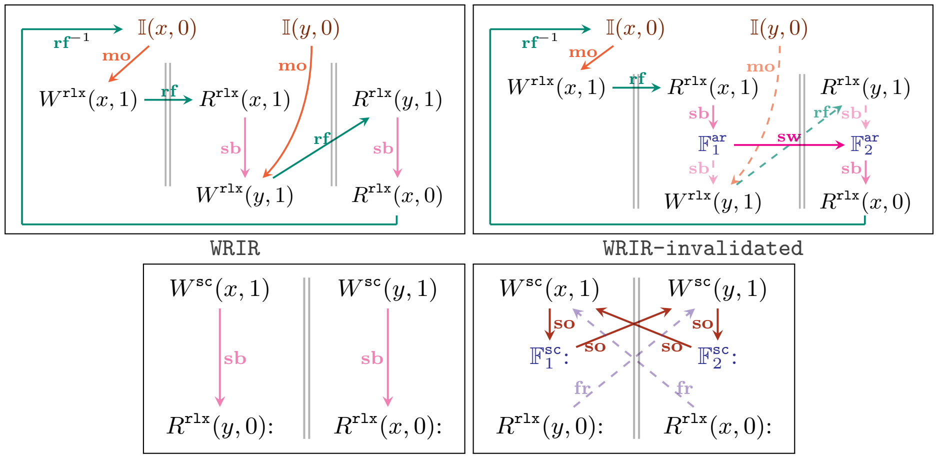
Additionally, *all* `sc` ordered events in a trace τ must be related by a total order ($\rightarrow_\tau^{\mathbf{to}}$) that concurs with the coherence conditions. We use an irreflexive relation called *from-reads* ($\rightarrow_\tau^{\mathbf{fr}} \triangleq \rightarrow_\tau^{\mathbf{rf}^{-1}}; \rightarrow_\tau^{\mathbf{mo}}$) for ordering reads with *later* writes. Consequently, $\rightarrow_\tau^{\mathbf{to}}$ must satisfy the following condition [12,25], referred to as (`to-sc`) and formally defined in the extended version [22].

- $\forall e_1^{\mathtt{sc}}, e_2^{\mathtt{sc}} \in \mathcal{E}_\tau^{(\mathtt{sc})}$ if $e_1^{\mathtt{sc}} \rightarrow_\tau^{\mathbf{to}} e_2^{\mathtt{sc}}$ then $(e_2^{\mathtt{sc}}, e_1^{\mathtt{sc}}) \notin \rightarrow_\tau^{\mathbf{hb}} \cup \rightarrow_\tau^{\mathbf{mo}} \cup \rightarrow_\tau^{\mathbf{rf}} \cup \rightarrow_\tau^{\mathbf{fr}}$; *and*,
- an `sc` read (or any read with an `sc` fence $\rightarrow_\tau^{\mathbf{sb}}$ ordered before it) must not read from an `sc` write that is not *immediately* $\rightarrow_\tau^{\mathbf{to}}$ ordered before it.

Conjunction of `(coherence conditions)` and `(to-sc)` forms the sufficient condition to determine if a trace τ is valid under *C11* (formally defined in [22]).

HB with *C11* fences. *C11* fences form $\rightarrow_\tau^{\mathbf{ithb}}$ with other events [9,12]. A fence can be associated with the memory orders `rel`, `acq`, `ar` and `sc`. An appropriately

[3] *release-sequence* of e_w in τ: maximal contiguous sub-sequence of $\rightarrow_\tau^{\mathbf{mo}}$ that starts at e_w and contains: (i) write events of $thr(e_w)$, (ii) rmw events of other threads [9,12].



`WRIR` `WRIR-invalidated` `SB` `SB-inv`

placed fence can form $\rightarrow_\tau^{\mathbf{sw}}$ and $\rightarrow_\tau^{\mathbf{dob}}$ relation from an $\rightarrow_\tau^{\mathbf{rf}}$ relation between events of different threads (formal described in the extended version [22]).

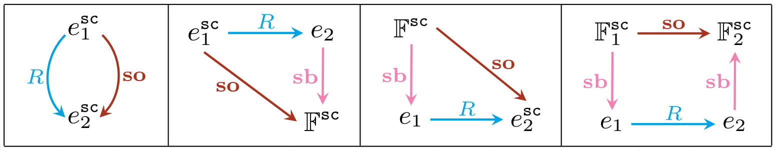
5 Invalidating buggy traces with *C11* fences

The key idea behind the proposed techniques is to introduce fences such that either `(coherence conditions)` or `(to-sc)` are violated. This section introduces two approaches for determining if the trace is rendered invalid with fences.

Consider $\tau^{\mathtt{imm}}$ of a buggy trace τ. The candidate fences of $\tau^{\mathtt{imm}}$ inflate $\rightarrow_\tau^{\mathbf{sb}}$, $\rightarrow_\tau^{\mathbf{sw}}$, $\rightarrow_\tau^{\mathbf{dob}}$ and $\rightarrow_\tau^{\mathbf{ithb}}$ relations (fences do not contribute to $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{mo}}$, $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{rf}}$ and $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{fr}}$). The inflated relations are denoted as $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{sb}}$, $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{sw}}$, $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{dob}}$ and $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{ithb}}$. We propose *Weak-FenSying* and *Strong-FenSying* to detect the invalidity of $\tau^{\mathtt{imm}}$.

Weak-FenSying. Weak-FenSying computes compositions of relations that correspond to the `(coherence conditions)`. It then checks if there exist cycles in the compositions (using *Johnson's algorithm* [13]). The approach assumes the memory order `ar` for all candidate fences. Consider a buggy trace `(WRIR)` where x and y have 0 as initial values. Weak-FenSying detects a cycle in $\rightarrow_\tau^{\mathbf{mo}};\rightarrow_\tau^{\mathbf{rf}};\rightarrow_\tau^{\mathbf{hb}};\rightarrow_\tau^{\mathbf{rf}^{-1}}$ with the addition of candidate fences $\mathbb{F}_1^{\mathtt{ar}}$ and $\mathbb{F}_2^{\mathtt{ar}}$ as shown in `(WRIR-invalidated)`. This violates the condition `(co-mrhi)`, thus, invalidating `(WRIR)`.

Strong-FenSying. This technique works with the assumption that all candidate fences have the order `sc`. Strong-FenSying detects the infeasibility in constructing a $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}}$ that adheres to `(to-sc)`. In order to detect violation of `(to-sc)`, Strong-FenSying introduces a possibly reflexive relation on `sc`-ordered events of $\tau^{\mathtt{imm}}$, called `sc`-*order* ($\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}}$). The $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}}$ relation is such that a total order cannot be formed on the `sc` events of $\tau^{\mathtt{imm}}$ iff a cycle exists in $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}}$. All `sc` event pairs ordered by $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{hb}}$, $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{mo}}$, $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{rf}}$ and $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{fr}}$ are contained in $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}}$. Notably, pairs of `sc` events that do not have a definite order are not ordered by $\rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}}$. This is because if such a pair of events is involved in a cycle then we can freely flip their order and eliminate the cycle. Consider the buggy trace `(SB)`, $W^{\mathtt{sc}}(x,1) \rightarrow^{\mathbf{to}} W^{\mathtt{sc}}(y,1)$ and $W^{\mathtt{sc}}(y,1) \rightarrow^{\mathbf{to}} W^{\mathtt{sc}}(x,1)$ are both valid total-orders



(soee) (soef) (sofe) (soff)

on the `sc` events of the trace. The set $\rightarrow^{\mathbf{so}}_{\tau^{\mathtt{imm}}}$ does not contain either of the two event pairs and would be empty for this example.

As a consequence, pairs of events that do not have definite total order cannot contribute to the reflexivity of $\rightarrow^{\mathbf{so}}_{\tau}$ and can be safely ignored. Thus, $\rightarrow^{\mathbf{so}+}_{\tau} \subseteq \rightarrow^{\mathbf{to}}_{\tau}$ for a trace τ. Further, if a total order cannot be formed on `sc` ordered events then a corresponding cycle exists in $\rightarrow^{\mathbf{so}}_{\tau^{\mathtt{imm}}}$. The observations are formally presented with supporting proofs in the extended version [22]).
Definition 3 formally presents $\rightarrow^{\mathbf{so}}_{\tau^{\mathtt{imm}}}$ based on the above stated considerations.

Definition 3. `sc`-order ($\rightarrow^{\mathbf{so}}_{\tau^{\mathtt{imm}}}$)
$\forall e_1, e_2 \in \mathcal{E}_\tau$ s.t. $(e_1, e_2) \in R$, where $R = \rightarrow^{\mathbf{hb}}_{\tau} \cup \rightarrow^{\mathbf{mo}}_{\tau} \cup \rightarrow^{\mathbf{rf}}_{\tau} \cup \rightarrow^{\mathbf{fr}}_{\tau}$

- if $e_1, e_2 \in \mathcal{E}^{(\mathtt{sc})}_{\tau}$ then $e_1 \rightarrow^{\mathbf{so}}_{\tau^{\mathtt{imm}}} e_2$; (soee)
- if $e_1 \in \mathcal{E}^{(\mathtt{sc})}_{\tau}$, $\exists \mathbb{F}^{\mathtt{sc}} \in \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_{\tau^{\mathtt{imm}}}$ s.t. $e_2 \rightarrow^{\mathbf{sb}}_{\tau^{\mathtt{imm}}} \mathbb{F}^{\mathtt{sc}}$ then $e_1 \rightarrow^{\mathbf{so}}_{\tau^{\mathtt{imm}}} \mathbb{F}^{\mathtt{sc}}$; (soef)
- if $e_2 \in \mathcal{E}^{(\mathtt{sc})}_{\tau}$, $\exists \mathbb{F}^{\mathtt{sc}} \in \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_{\tau^{\mathtt{imm}}}$ s.t. $\mathbb{F}^{\mathtt{sc}} \rightarrow^{\mathbf{sb}}_{\tau^{\mathtt{imm}}} e_1$ then $\mathbb{F}^{\mathtt{sc}} \rightarrow^{\mathbf{so}}_{\tau^{\mathtt{imm}}} e_2$; (sofe)
- if $\exists \mathbb{F}_1^{\mathtt{sc}}, \mathbb{F}_2^{\mathtt{sc}} \in \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_{\tau^{\mathtt{imm}}}$ s.t. $\mathbb{F}_1^{\mathtt{sc}} \rightarrow^{\mathbf{sb}}_{\tau^{\mathtt{imm}}} e_1$ and $e_2 \rightarrow^{\mathbf{sb}}_{\tau^{\mathtt{imm}}} \mathbb{F}_2^{\mathtt{sc}}$ then $\mathbb{F}_1^{\mathtt{sc}} \rightarrow^{\mathbf{so}}_{\tau^{\mathtt{imm}}} \mathbb{F}_2^{\mathtt{sc}}$. (soff)

The trace depicted in (SB) can be invalidated with strong fences as shown in (SB-inv). The `sc` events of (SB-inv) cannot be totally ordered and Strong-FenSying detects the same through a cycle in $\rightarrow^{\mathbf{so}}$ (formed by (soee) and (sofe)).
Scope of `FenSying`/`fFenSying`. Our work synthesizes *C11* fences and stands fundamentally different from techniques that strengthen the memory orders of events. On the one hand, `sc` fences cannot restore sequential consistency; thus, strengthening memory orders may invalidate buggy traces that the strongest *C11* fences cannot. On the other hand, strengthening may lead to sub-optimal byte-code. The difference is explained in the extended version [22].

6 Methodology

Buggy traces and candidate fences. Algorithms 1 and 2 present `FenSying` and `fFenSying`, respectively. The algorithms rely on an external buggy trace generator (`BTG`) for the buggy trace(s) of P (lines 2,8). The candidate fences are inserted (to obtain $\tau^{\mathtt{imm}}$), and the event relations are updated (lines 16-17).
Detecting violation of trace coherence. The algorithms detect possible violations of trace coherence conditions resulting from the candidate fences at lines 18-19 of the function `synthesisCore`. Figures (RWRW-inv-sync) and (RWRW-inv-sync-opt) represent two instances of violations of (co-rh) detected by Weak-FenSying (through a cycle $W^{\mathtt{rlx}}(y,1) \rightarrow^{\mathbf{rf}}_{\tau^{\mathtt{imm}}} R^{\mathtt{rlx}}(y,1) \rightarrow^{\mathbf{hb}}_{\tau^{\mathtt{imm}}} W^{\mathtt{rlx}}(y,1)$). The candidate solutions corresponding to these cycles (which include only candidate fences) are $\{\mathbb{F}_{12}, \mathbb{F}_{22}\}$ and $\{\mathbb{F}_{12}\}$. Further, for the same example, (RWRW-inv-to)

Algorithm 1: FenSying (P)

1 $\Phi := \top$; $\mathcal{C} := \emptyset$
2 **forall** $\tau \in$ buggyTraces(P) **do**
3 $\quad \Phi_\tau, \mathcal{C}_\tau :=$ synthesisCore(τ)
4 $\quad \Phi := \Phi \wedge \Phi_\tau$; $\mathcal{C} := \mathcal{C} \cup \mathcal{C}_\tau$
5 minΦ := minModel(Φ)
6 $\mathcal{F}$:= assignMO(minΦ, $\mathcal{C}$)
7 **return** syn$(P, \mathcal{F})$

Algorithm 2: fFenSying (P)

8 **if** $\exists \tau \in$ buggyTraces(P) **then**
9 $\quad \Phi, \mathcal{C} :=$ synthesisCore(τ)
10 $\quad$ minΦ := minModel(Φ)
11 $\quad \mathcal{F}$:= assignMO(minΦ, $\mathcal{C}$)
12 $\quad P' :=$ syn$(P, \mathcal{F})$
13 $\quad$ **return** fFenSying (P')
14 **else return** P

15 **Function** synthesisCore(τ) /* $\tau = \langle \mathcal{E}_\tau, \xrightarrow{\text{hb}}_\tau, \xrightarrow{\text{mo}}_\tau, \xrightarrow{\text{rf}}_\tau \rangle$ */:
16 $\quad \mathcal{E}_{\tau^{\text{imm}}} := \mathcal{E}_\tau \cup$ candidateFences(τ)
17 $\quad (\xrightarrow{\text{hb}}_{\tau^{\text{imm}}}, \xrightarrow{\text{mo}}_{\tau^{\text{imm}}}, \xrightarrow{\text{rf}}_{\tau^{\text{imm}}}, \xrightarrow{\text{rf}^{-1}}_{\tau^{\text{imm}}}, \xrightarrow{\text{fr}}_{\tau^{\text{imm}}}) :=$ computeRelations$(\tau, \mathcal{E}_{\tau^{\text{imm}}})$
18 $\quad$ weakCycles$_\tau$:= weakFensying(τ^{imm})
19 $\quad$ strongCycles$_\tau$:= strongFensying(τ^{imm})
20 $\quad$ **if** weakCycles$_\tau = \emptyset \wedge$ strongCycles$_\tau = \emptyset$ **then**
21 $\quad\quad$ **return** /* ABORT: cannot stop τ with *C11* fences */
22 $\quad \Phi_\tau := \mathcal{Q}($weakCycles$_\tau \vee$ strongCycles$_\tau$); $\mathcal{C}_\tau :=$weakCycles$_\tau \cup$ strongCycles$_\tau$
23 $\quad$ **return** $\Phi_\tau, \mathcal{C}_\tau$

represents a violation detected by Strong-FenSying with the candidate solution $\{\mathbb{F}_{12}, \mathbb{F}_{22}\}$. The algorithms discard all candidate fences other than $\mathbb{F}_{12}$ and $\mathbb{F}_{22}$ from future considerations (assuming no other violations were detected). Now τ can be invalidated as the set of cycles is nonempty (line 20).

The complexity of detecting all cycles for a trace is $\mathcal{O}((|\mathcal{E}_\tau|+\mathtt{E}).(\mathtt{C}+1))$ where $\mathtt{C}$ represents the number of cycles of τ and $\mathtt{E}$ represents the number of pairs of events in $\mathcal{E}_\tau$. Note that $\mathtt{E}$ is in $O(|\mathcal{E}_\tau|^2)$ and $\mathtt{C}$ is in $O(|\mathcal{E}_\tau|!)$. Thus, Weak- and Strong-Fensying have exponential complexities in the number of traces and the number of events per trace.

Reduction for optimality. The algorithms use a SAT solver to determine the optimal number of candidate fences. The candidate fences from each candidate solution of τ are conjuncted to form a SAT query. Further, to retain at least one solution corresponding to τ the algorithms take a disjunction of the conjuncts. The SAT query is represented in the algorithm as $\Phi_\tau := \mathcal{Q}(\texttt{weakCycles}_\tau \vee \texttt{strongCycles}_\tau)$ (line 22) and presented in Equation 1 (where $\mathbf{W}_\tau$ and $\mathbf{S}_\tau$ represent $\texttt{weakCycles}_\tau$ and $\texttt{strongCycles}_\tau$ and $\mathtt{W}^{\mathbb{F}}$ and $\mathtt{S}^{\mathbb{F}}$ represent the set of candidate fences in cycles $\mathtt{W}$ and $\mathtt{S}$ respectively). Further, FenSying combines the SAT formulas corresponding to each buggy trace via conjunction (line 4), shown in Equation 2. However, note that for fFenSying $\Phi = \Phi_\tau$.

$$\Phi_\tau = \Big(\bigvee_{\mathtt{W} \in \mathbf{W}_\tau} \bigwedge_{\mathbb{F}_\mathtt{w} \in \mathtt{W}^{\mathbb{F}}} \mathbb{F}_\mathtt{w}\Big) \vee \Big(\bigvee_{\mathtt{S} \in \mathbf{S}_\tau} \bigwedge_{\mathbb{F}_\mathtt{s} \in \mathtt{S}^{\mathbb{F}}} \mathbb{F}_\mathtt{s}\Big) \quad (1) \qquad \Phi = \bigwedge_{\tau \in \mathtt{BT}} \Phi_\tau \quad (2)$$

We use a SAT solver to compute the *min-model* (minΦ) of the query Φ (lines 5,10). For instance, the query for (RWRW-bt) is $\Phi = (\mathbb{F}_{12}) \vee (\mathbb{F}_{12} \wedge \mathbb{F}_{22}) \vee (\mathbb{F}_{12} \wedge \mathbb{F}_{22})$ and min-model, min$\Phi = \{\mathbb{F}_{12}\}$. The solution to the SAT query returns the smallest set of fences to be synthesized.

The complexity of constructing the query Φ_τ is $\mathcal{O}(\texttt{C.F})$, where `C` is the number of cycles per trace and `F` is the number of fences per cycle. The structure of the query Φ corresponds to the *Head-cycle-free* (HCF) class of `CNF` theories; hence, the min-model computation falls in the `FP` complexity class [8].

Determining optimal memory orders of fences. The set `min`Φ gives a sound solution that is optimal only in the number of fences. The function `assignMO` (lines 6,11) assigns the weakest memory order to the fences in `min`Φ that is sound. Let `min-cycles` represent a set of cycles such that every candidate fence in the cycles belongs to `min`Φ. The `assignMO` function computes memory order for fences of `min-cycles` of each trace as follows: If a cycle $c \in$ `min-cycles` is detected, then its fences must form a $\rightarrow^{\mathbf{sw}}_{\tau^{\texttt{imm}}}$ or $\rightarrow^{\mathbf{dob}}_{\tau^{\texttt{imm}}}$ with an event of $\tau^{\texttt{imm}}$ (since, candidate fences only modify $\rightarrow^{\mathsf{sb}}_{\tau}$, $\rightarrow^{\mathbf{sw}}_{\tau}$ and $\rightarrow^{\mathbf{dob}}_{\tau}$). Let $R = \rightarrow^{\mathbf{sw}}_{\tau^{\texttt{imm}}} \cup \rightarrow^{\mathbf{dob}}_{\tau^{\texttt{imm}}}$. The scheme to compute fence types is as follows:

- If a fence $\mathbb{F}$ in a weak cycle c is related to an event e of c by R as $eR\mathbb{F}$, then $\mathbb{F}$ is assigned the memory order `acq`;
- if an event e in c is related to $\mathbb{F}$ as $\mathbb{F}Re$ then $\mathbb{F}$ is assigned `rel`;
- if events e, e' of c are related to $\mathbb{F}$ as $eR\mathbb{F}Re'$ then $\mathbb{F}$ is assigned `ar`.
- All the fences in a strong cycle are assigned the memory order `sc`.

Consider a cycle c: $e \rightarrow^{\mathsf{sb}}_{\tau'^{\texttt{imm}}} \mathbb{F}_1 \rightarrow^{\mathbf{sw}}_{\tau'^{\texttt{imm}}} \mathbb{F}_2 \rightarrow^{\mathbf{sw}}_{\tau'^{\texttt{imm}}} \mathbb{F}_3 \rightarrow^{\mathsf{sb}}_{\tau'^{\texttt{imm}}} e' \rightarrow^{\mathbf{rf}}_{\tau'^{\texttt{imm}}} e$ representing a violation of $\rightarrow^{\mathbf{rf}}_{\tau'^{\texttt{imm}}}; \rightarrow^{\mathbf{hb}}_{\tau'^{\texttt{imm}}}$ irreflexivity (condition `(co-rh)`). According to the scheme discussed above, the fences $\mathbb{F}_1$, $\mathbb{F}_2$ and $\mathbb{F}_3$ are assigned the memory orders `rel`, `ar` and `acq` respectively and $wt(c) = 4$ (defined in §3).

Further, `assignMO` iterates over all buggy traces and detects the sound weakest memory order for each fence across all traces as follows. Assume a cycle c_1 in ${\tau_1}^{\texttt{imm}}$ and a cycle c_2 in ${\tau_2}^{\texttt{imm}}$. The function computes a union of the fences of τ_1 and τ_2 while choosing the stronger memory order for each fence that is present in both the cycles. In doing so, both τ_1 and τ_2 are invalidated. Further, when two candidate solutions have the same set of fences, the function selects the one with the lower weight.

Consider the cycles of buggy traces τ_1 and τ_2 shown in `(candidate-fences)`. Let `min`$\Phi = \{\mathbb{F}_1, \mathbb{F}_2, \mathbb{F}_3\}$. The memory orders of the fences for each trace are shown with superscripts and the weights of the cycle $\tau_1 c_1$, $\tau_1 c_2$ and $\tau_2 c_1$ are written against the name of the cycles. The candidate solutions $\tau_1 c_1$ and $\tau_1 c_2$ are combined with $\tau_2 c_1$ to form $\tau_{12} c_{11}$ and $\tau_{12} c_{21}$ of weights 5 and 4, respectively. The solution $\tau_{12} c_{11}$ is of higher weight and is discarded. In $\tau_{12} c_{21}$, the optimal memory orders `rel`, `acq` and `ar` are assigned to fences $\mathbb{F}_1$, $\mathbb{F}_2$ and $\mathbb{F}_3$, respectively. It is possible that `min-cycles` may contain fences originally in P. If the process discussed above computes a stronger memory order for a program fence than its original order in P, then the technique strengthens the memory order of the fence to the computed order. Note that this reasoning across traces does not occur in `fFenSying` as it considers only one trace at a time.

Determining the optimal memory orders has a complexity in $\mathcal{O}(\texttt{BT.C.F}+\texttt{M}^{\texttt{BT}})$, where `BT` if the number of buggy traces of P, `C` and `F` are defined as before, and `M` is the number of min-cycles per trace.

$\tau_1 c_1(4)$: $\mathbb{F}_1^{\mathtt{ar}} \wedge \mathbb{F}_2^{\mathtt{ar}}$ $\tau_1 c_2(4)$: $\mathbb{F}_1^{\mathtt{rel}} \wedge \mathbb{F}_2^{\mathtt{acq}} \wedge \mathbb{F}_3^{\mathtt{ar}}$ cycles of τ_1
$\tau_2 c_1(3)$: $\mathbb{F}_1^{\mathtt{rel}} \wedge \mathbb{F}_2^{\mathtt{acq}} \wedge \mathbb{F}_3^{\mathtt{acq}}$ cycle of τ_2
$\tau_{12} c_{11}(5)$: $\mathbb{F}_1^{\mathtt{ar}} \wedge \mathbb{F}_2^{\mathtt{ar}} \wedge \mathbb{F}_3^{\mathtt{acq}}$ $\tau_{12} c_{21}(4)$: $\mathbb{F}_1^{\mathtt{rel}} \wedge \mathbb{F}_2^{\mathtt{acq}} \wedge \mathbb{F}_3^{\mathtt{ar}}$

candidate-fences

cycles in τ_1 ($\mathtt{C}_{\tau_1}$): $\{\mathbb{F}_1, \mathbb{F}_2, e_1\}$ and $\{\mathbb{F}_1, \mathbb{F}_3, \mathbb{F}_4\}$ $\Phi_{\tau_1} = (\mathbb{F}_1 \wedge \mathbb{F}_2) \vee (\mathbb{F}_1 \wedge \mathbb{F}_3 \wedge \mathbb{F}_4)$ cycles in τ_2 ($\mathtt{C}_{\tau_2}$): $\{\mathbb{F}_3, \mathbb{F}_4\}$ $\Phi_{\tau_2} = (\mathbb{F}_3 \wedge \mathbb{F}_4)$

3-fence

In our experimental observation (refer to §7), the number of buggy traces analyzed by `fFenSying` is significantly less than $|\mathtt{BT}|$. Therefore, in practice, the complexity of various steps of `fFenSying` that are dependent on `BT` reduces exponentially by a factor of $|\mathtt{BT}|$.

Nonoptimality of `fFenSying`. Consider the example (`3-fence`). It shows cycles in two buggy traces τ_1 and τ_2 of an input program. `FenSying` provides the formula $\Phi_{\tau_1} \wedge \Phi_{\tau_2}$ to the `SAT` solver and the optimal solution obtained is $(\mathbb{F}_1 \wedge \mathbb{F}_3 \wedge \mathbb{F}_4)$. However, `fFenSying` considers the formula Φ_{τ_1} and Φ_{τ_2} in separate iterations and may return a nonoptimal result $(\mathbb{F}_1 \wedge \mathbb{F}_2) \wedge (\mathbb{F}_3 \wedge \mathbb{F}_4)$.

We prove the soundness of `fFenSying` and `FenSying` with Theorems 1 and 2 respectively and the optimality of `FenSying` with Theorem 3. The theorems are formally presented with proofs in the extended version [22].

Theorem 1. *`fFenSying` is sound. If a buggy trace τ of P can be invalidated using C11 fences then `fFenSying` will invalidate τ.*

Theorem 2. *`FenSying` is sound. If a buggy program P can be fixed using C11 fences, then `FenSying` will invalidate all buggy traces of P.*

Theorem 3. *`FenSying` is optimal. `FenSying` synthesizes optimal number of fences with optimal memory orders.*

7 Implementation and Results

Implementation details. The techniques are implemented in `Python`. Weak-FenSying and Strong-FenSying use *Johnson's* cycle detection algorithm in the *networkx* library. We use `Z3` theorem prover to find the *min-model* of `SAT` queries. As a `BTG`, we use CDSChecker [20], an open-source model checker, for the following reasons;

1. CDSChecker supports the *C11* semantics. Most other techniques are designed for a variant [15] or subset [1,3,23] of *C11*.
2. CDSChecker returns buggy traces along with the corresponding $\rightarrow_{\tau}^{\mathbf{hb}}$, $\rightarrow_{\tau}^{\mathbf{rf}}$ and $\rightarrow_{\tau}^{\mathbf{mo}}$ relations.
3. CDSChecker does not halt at the detection of the first buggy trace; instead, it continues to provide all buggy traces as required by `FenSying`.

To bridge the gap between CDSChecker's output and our requirements, we modify CDSChecker's code to accept program location as an attribute of the program events and to halt at the first buggy trace when specified. `FenSying` and

Table 1. Litmus Testing Summary

Litmus Tests Summary

Tests	min-BT	max-BT	avg-BT	min-syn	max-syn	avg-syn	min-str	max-str	avg-str
1389	1	9	1.05	1	4	2.25	0	0	0

BT: #buggy traces, syn: #fences synthesized, str: # fences strengthened
min: minimum, max: maximum, avg: average

Results Summary

	completed (syn+no fix)	TO	NO	Tbtg (total)	TF (total)	Ttotal
FenSying	1333 (1185+148)	56	0	50453.19	36896.06	87266.09
fFenSying	1355 (1207+148)	34	0	30703.71	49068.61	79772.32

Times in seconds. TO: 15min for BTG + 15min for technique
Tbtg: Time of BTG, TF: Time of FenSying or fFenSying, Ttotal: Tbtg+TF

fFenSying are available as an open-source tool that performs fence synthesis for *C11* programs at: https://github.com/singhsanjana/fensying.

Experimental setup. The experiments were performed on an Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz with 32GB RAM and 32 cores. We collected a set of 1389 litmus tests of buggy *C11* input programs (borrowed from Tracer [3]) to validate the correctness of FenSying and fFenSying experimentally. We study the performance of FenSying and fFenSying on a set of benchmarks borrowed from previous works on model checking under *C11* and its variants [1,3,20,23].

Experimental validation. The summary of the 1389 litmus tests is shown under *Litmus Tests Summary*, Table 1. The number of buggy traces for the litmus tests ranged between 1-9 with an average of 1.05, while the number of fences synthesized ranged between 2-4. None of the litmus tests contained fences in the input program. Hence, no fences were strengthened in any of the tests.

We present the results of FenSying and fFenSying under *Result Summary*, Table 1. The results have been averaged over five runs for each test. fFenSying timed out (column 'TO') on a fewer number of tests (34 tests) in comparison to FenSying (56 tests). The techniques could not fix 148 tests with *C11* fences ('no fix'). The column 'NO' represents the number of tests where the fences synthesized or strengthened is nonoptimal. To report the values of 'NO', we conducted a sanity test on the fixed program as follows: we create versions $P_1, ...P_k$ of the fixed program $P^{\texttt{fx}}$ s.t. in each version, one of the fences of $P^{\texttt{fx}}$ is either weakened or eliminated. Each version is then tested separately on BTG. The sanity check is successful if a buggy trace is returned for each version.

Performance analysis. We contrast the performance of the techniques using a set of benchmarks that produce buggy traces under *C11*. The results are averaged over five runs. Table 2 reports the results where '#BT' shows the number of buggy traces, 'iter' shows the minimum:maximum number of iterations performed by fFenSying over the five runs and, 'FTo' and 'BTo' represent FenSying/fFenSying time-out and BTG time-out, respectively (set to 15 minutes each). A '?' in '#BT' signifies that BTG could not scale for the test, so the number of buggy traces is unknown. The column ('syn+str') under fFenSying reports the minimum:maximum number of fences synthesized and/or strengthened. We

Table 2. Comparative performance analysis

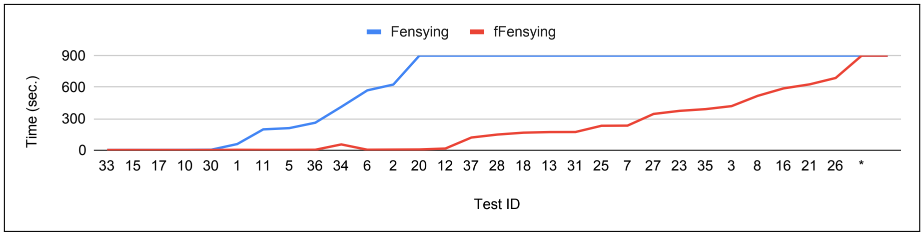
			FenSying				fFenSying				
Id	Name	#BT	syn+str	Tbtg	TF	Ttotal	iter	syn+str	Tbtg	TF	Ttotal
1	peterson(2,2)	30	1+0	2.63	54.31	56.94	1:1	1:1+0:0	0.18	2.07	2.25
2	peterson(2,3)	198	1+0	29.96	594.34	624.3	1:1	1:1+0:0	0.53	3.58	4.11
3	peterson(4,5)	?	–	–	FTo	–	1:1	1:1+0:0	397.51	21.07	418.58
4	peterson(5,5)	?	–	BTo	–	–	1:1	1:1+0:0	BTo	31.52	*931.52
5	barrier(5)	136	1+0	1.09	207.74	208.83	1:1	1:1+0:0	0.13	1.40	1.53
6	barrier(10)	416	1+0	3.37	565.44	568.81	1:1	1:1+0:0	0.2	2.70	2.9
7	barrier(100)	31106	–	–	FTo	–	1:1	1:1+0:0	34.2	198.54	232.74
8	barrier(150)	?	–	–	FTo	–	1:1	1:1+0:0	117.09	399.20	516.29
9	barrier(200)	–	–	–	–	–	–	–	–	FTo	–
10	store-buffer(2)	6	2+0	0.08	0.91	0.99	1:1	2:2+0:0	0.04	0.05	0.09
11	store-buffer(4)	20	2+0	1.61	195.35	196.96	1:1	2:2+0:0	1.20	0.05	1.25
12	store-buffer(5)	30	–	–	FTo	–	1:1	2:2+0:0	14.07	0.22	14.29
13	store-buffer(6)	42	–	–	FTo	–	1:1	2:2+0:0	171.09	0.15	171.24
14	store-buffer(10)	?	–	BTo	–	–	1:1	2:2+0:0	BTo	0.05	*900.05
15	dekker(2)	54	2+0	0.17	0.27	0.44	1:1	2:2+0:0	0.26	0.04	0.3
16	dekker(3)	1596	–	–	FTo	–	1:1	2:2+0:0	586.46	1.34	587.8
17	dekker-fen(2,3)	54	1+1	0.15	0.29	0.44	1:1	1:1+1:1	0.25	0.05	0.3
18	dekker-fen(3,2)	730	–	–	FTo	–	1:1	1:1+1:1	159.84	5.56	165.4
19	dekker-fen(3,4)	3076	–	BTo	–	–	1:1	1:1+1:1	BTo	6.06	*906.06
20	burns(1)	36	–	–	FTo	–	7:8	8:10+2:2	0.61	4.69	5.3
21	burns(2)	10150	–	–	FTo	–	6:7	8:10+0:1	71.53	554.6	626.13
22	burns(3)	?	–	BTo	–	–	–	–	–	FTo	–
23	burns-fen(2)	100708	–	–	FTo	–	5:7	4:6+3:3	329.41	43.96	373.37
24	burns-fen(3)	?	–	BTo	–	–	5:7	4:6+3:3	BTo	70.14	*970.14
25	linuxrwlocks(2,1)	10	–	–	FTo	–	1:1	2:2+0:0	0.13	0.12	0.25
26	linuxrwlocks(3,8)	353	–	–	FTo	–	2:2	3:4+0:0	686.52	0.41	*686.93
27	seqlock(2,1,2)	500	–	–	FTo	–	1:1	1:1+0:0	341.54	2.38	343.92
28	seqlock(1,2,2)	592	–	–	FTo	–	1:2	1:2+0:0	119.88	27.69	147.57
29	seqlock(2,2,3)	?	–	BTo	–	–	1:2	1:2+0:0	BTo	88.52	988.52*
30	bakery(2,1)	6	1+0	0.25	25.42	2.88	1:1	1:1+0:0	0.07	0.18	0.25
31	bakery(4,3)	7272	–	–	FTo	–	1:1	1:1+0:0	166.11	5.68	171.79
32	bakery(4,4)	50402	–	–	FTo	–	1:1	1:1+0:0	BTo	18.17	918.17*
33	lamport(1,1,2)	1	No fix.	0.06	0.05	0.11	1:1	No fix.	0.04	0.05	0.09
34	lamport(2,2,1)	1	No fix.	411.94	0.05	411.99	1:1	No fix.	53.34	0.05	53.39
35	lamport(2,2,3)	?	–	BTo	–	–	1:1	No fix.	389.77	0.05	389.82
36	flipper(5)	297	2+0	6.22	254.18	260.40	1:1	2+0	2.51	0.02	2.53
37	flipper(7)	4493	–	–	FTo	–	1:1	2+0	119.21	0.02	119.23
38	flipper(10)	?	–	–	FTo	–	1:1	2+0	BTo	0.03	900.03*

Tbtg: Time of BTG, TF: Time of technique (FenSying or fFenSying), Ttotal: Tbtg+TF

add a '*' against the time when BTG timed out in detecting that the fixed program has no more buggy traces.

The performance of FenSying and fFenSying is diagrammatically contrasted in Figure 1. It is notable that fFenSying significantly outperforms FenSying in terms of the time of execution and scalability and adds extra fences in only 7 tests with an average of 1.57 additional fences. With the increase in the number of buggy traces, an exponential rise in FenSying's time leading to FTo was observed; except in cases 12, 13, 20, and 25, where FenSying times out with as low as 10 traces. The tests time-out in *Johnson's* cycle detection due to a high density of the number of related events or the number of cycles.

fFenSying analyzes a remarkably smaller number of buggy traces ('iter') in comparison with '#BT' ($\leq$2 traces for $\sim$85% of tests). We conclude that a solution corresponding to a single buggy trace fixes more than one buggy traces. As a result, fFenSying can scale to tests with thousands of buggy traces and



⋆ represents the remaining Test IDs (tests that timeout for both `FenSying` and `fFenSying`)

Fig. 1. Performance comparison between `FenSying` and `fFenSying`

we witness an average speedup of over 67x, with over 100x speedup in ∼41% of tests, against `FenSying`.

Interesting cases. Consider test 16, where `BTG` times out in 3/5 runs and completes in ∼100s in the remaining 2 runs. A fence is synthesized between two events, e_1 and e_2, that are inside a loop. Additionally, e_1 is within a condition. Depending on where the fence is synthesized (within the condition or outside it), `BTG` either runs out of time or finishes quickly. Similarly, `BTG` for test 26 times out in 3/5 runs. However, the reason here is the additional nonoptimal fences synthesized that increase the analysis overhead of the chosen `BTG` (CDSChecker).

Note that, for most benchmarks, `fFenSying`'s scalability is limited by BTo and observably `fFenSying`'s time is much lesser than FTo for such cases. Therefore, an alternative `BTG` would significantly improve `fFenSying`'s performance.

8 Related Work

The literature on fence synthesis is rich with techniques targeting the x86-TSO [2,5,6,7,10] and sparc-PSO [4,18] memory models or both [14,16,19]. The work in [10] and [16] perform fence synthesis for ARMv7 and RMO memory models. The works in [4,7,11] are proposed for Power memory model, where [11] also supports IA-32 memory model.

Most fence synthesis techniques introduce additional ordering in the program events with the help of fences [5,6,7,10,11,14,18,19,24]. However, the axiomatic definition of ordering varies with memory models. As a consequence, most existing techniques (such as those for TSO and PSO) may not detect *C11* buggy traces due to a strong implicit ordering. While the techniques [6,7,24] are parametric in or oblivious to the memory model, they introduce ordering between *pairs* of events that is *globally visible* (to all threads). Such an ordering constraint is restrictive for weaker models such as *C11* that may require ordering on a *set* of events that may be *conditionally visible* to a thread. Similarly, [14] proposes a bounded technique applicable to any memory model that supports interleaving with reordering. Program outcomes under *C11* may not be feasible under such a model. Moreover, any existing technique, cannot fix a *C11* input program while conserving its portability.

Some earlier works such as [2,7,11] synthesize fences to restrict outcomes to SC or its variant for store-buffering [5]. Most fence synthesis techniques

[4,14,16,18,19] attempt to remove traces violating a safety property specification under their respective axiomatic definition of memory model. Various works [4,5,10,14,16,19,24] perform optimal fence synthesis where the optimality (in the absence of types of fences) is simply defined as the smallest set of fences. Technique [6] assigns weights to various types of fences (similar to our work) and defines optimality on the summation of fence weights of candidate solutions. However, their definition of optimality is incomparable with ours, and no prior work establishes the advantage of one definition over the other.

Lastly, a recent technique [21] fixes a buggy *C11* program by strengthening memory access events instead of synthesizing fences.

9 Conclusion and Future Work

This paper proposed the first fence synthesis techniques for *C11* programs: an optimal (`FenSying`) and a near-optimal (`fFenSying`). The work also presented theoretical arguments that showed the correctness of the synthesis techniques. The experimental validation demonstrated the effectiveness of `fFenSying` vis-à-vis optimal `FenSying`. As part of future work, we will investigate extending the presented methods (i) to support richer constructs such as locks and (ii) to include strengthening memory accesses to fix buggy traces.

References

1. Abdulla, P.A., Arora, J., Atig, M.F., Krishna, S.: Verification of programs under the release-acquire semantics. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1117–1132 (2019)
2. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Automatic fence insertion in integer programs via predicate abstraction. In: International Static Analysis Symposium. pp. 164–180. Springer (2012)
3. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proceedings of the ACM on Programming Languages 2(OOPSLA), 1–29 (2018)
4. Abdulla, P.A., Atig, M.F., Lång, M., Ngo, T.P.: Precise and sound automatic fence insertion procedure under pso. In: International Conference on Networked Systems. pp. 32–47. Springer (2015)
5. Abdulla, P.A., Atig, M.F., Ngo, T.P.: The best of both worlds: Trading efficiency and optimality in fence insertion for tso. In: European Symposium on Programming Languages and Systems. pp. 308–332. Springer (2015)
6. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence. In: International Conference on Computer Aided Verification. pp. 508–524. Springer (2014)
7. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models. In: International Conference on Computer Aided Verification. pp. 258–272. Springer (2010)
8. Angiulli, F., Ben-Eliyahu-Zohary, R., Fassetti, F., Palopoli, L.: On the tractability of minimal model computation for some cnf theories. Artificial Intelligence 210, 56–77 (2014)

9. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing c++ concurrency. ACM SIGPLAN Notices 46(1), 55–66 (2011)
10. Bender, J., Lesani, M., Palsberg, J.: Declarative fence insertion. ACM SIGPLAN Notices 50(10), 367–385 (2015)
11. Fang, X., Lee, J., Midkiff, S.P.: Automatic fence insertion for shared memory multiprocessing. In: Proceedings of the 17th annual international conference on Supercomputing. pp. 285–294 (2003)
12. ISO/IEC-JTC1/SC22/WG21: Programming languages — C++ (2013), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf
13. Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM Journal on Computing 4(1), 77–84 (1975)
14. Joshi, S., Kroening, D.: Property-driven fence insertion using reorder bounded model checking. In: International Symposium on Formal Methods. pp. 291–307. Springer (2015)
15. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 96–110 (2019)
16. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. ACM SIGACT News 43(2), 108–123 (2012)
17. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in c/c++ 11. vol. 52, pp. 618–632. ACM New York, NY, USA (2017)
18. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in pso memory systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 339–353. Springer (2013)
19. Meshman, Y., Dan, A., Vechev, M., Yahav, E.: Synthesis of memory fences via refinement propagation. In: International Static Analysis Symposium. pp. 237–252. Springer (2014)
20. Norris, B., Demsky, B.: Cdschecker: checking concurrent data structures written with c/c++ atomics. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications. pp. 131–150 (2013)
21. Oberhauser, J., Chehab, R.L.d.L., Behrens, D., Fu, M., Paolillo, A., Oberhauser, L., Bhat, K., Wen, Y., Chen, H., Kim, J., et al.: Vsync: push-button verification and optimization for synchronization primitives on weak memory models. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 530–545 (2021)
22. Singh, S., Sharma, D., Jaju, I., Sharma, S.: Fence synthesis under the c11 memory model. arXiv preprint arXiv:2208.00285 (2022)
23. Singh, S., Sharma, D., Sharma, S.: Dynamic verification of c11 concurrency over multi copy atomics. In: 2021 International Symposium on Theoretical Aspects of Software Engineering (TASE). pp. 39–46. IEEE (2021)
24. Taheri, M., Pourdamghani, A., Lesani, M.: Polynomial-time fence insertion for structured programs. In: 33rd International Symposium on Distributed Computing (DISC 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
25. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common compiler optimisations are invalid in the c11 memory model and what we can do about it. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 209–220 (2015)