Effective Abstractions for Verification under Relaxed Memory Models

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

[1] ETH Zurich
[2] Technion

Abstract. We present a new abstract interpretation based approach for automatically verifying concurrent programs running on relaxed memory models.
Our approach is based on three key insights: (i) behaviors of relaxed models (e.g. TSO and PSO) are naturally captured using explicit encodings of *store buffers*. Directly using such encodings for program analysis is challenging due to *shift operations* on buffer contents that result in significant loss of analysis precision. We present a new abstraction of the memory model that eliminates expensive shifting of store buffer contents and significantly improves the precision and scalability of program analysis, (ii) an encoding of store buffer sizes that leverages knowledge of the abstract interpretation domain, further improving analysis precision, and (iii) a source-to-source transformation that realizes the above two techniques: given a program P and a relaxed memory model M, it produces a new program P_M where the behaviors of P running on M are over-approximated by the behavior of P_M running on sequential consistency (SC). This step makes it possible to directly use state-of-the-art analyzers under SC.
We implemented our approach and evaluated it on a set of finite and infinite-state concurrent algorithms under two memory models: Intel's x86 TSO and PSO. Experimental results indicate that our technique achieves better precision and efficiency than prior work: we can automatically verify algorithms with fewer fences, faster and with lower memory consumption.

1 Introduction

To improve performance, modern hardware architectures support relaxed memory models. A relaxed memory model allows the underlying architecture to reorder memory operations and execute them non-atomically. As a result, a concurrent program can have additional behaviors that would not be possible to obtain under the intuitive, sequentially consistent setting [16]. These additional relaxed behaviors complicate the task of reasoning about the correctness of the program, manually and automatically.

This necessitates the development of new, scalable and precise analysis techniques for automatic verification of (potentially infinite-state) concurrent programs running on relaxed memory models. Automatic verification in this setting is a challenging problem as the relaxed memory model can significantly increase the number and diversity of new behaviors, which in turn affects the overall precision and scalability of the analysis.

Our Approach We present a new analysis system for verifying concurrent programs running on relaxed memory models such as Intel's x86 TSO and PSO buffered memory models. Our system builds upon three core concepts:

First, we present a new abstraction that eliminates some of the expensive work in managing the store buffers required by the memory model, thus significantly reducing the analysis effort and improving its precision. This abstraction is also directly applicable and useful for other verification frameworks, both finite and infinite-state (e.g., bounded model checking, abstract interpretation and predicate abstraction).

Second, we show how to leverage knowledge of the particular program analysis used in this work (abstract interpretation with numerical domains) by encoding the size of the store buffers in a way that reduces the loss of precision under that abstract domain.

Third, we address the problem of building a robust analyzer that incorporates the above two concepts. We present a source-to-source transformation that enables direct reuse of program analyzers under sequential consistency for verifying concurrent programs running on relaxed memory models. That is, given a program P, a specification S and a memory model M, the transformation automatically constructs a new program P_M such that if $P_M \models_{SC} S$ then $P \models_M S$. The program P_M contains *an abstraction of the relaxed behaviors induced by* M, thereby ensuring soundness of the approach.

While prior works [10, 3, 18] also suggest source-to-source transformations, we show experimentally that our approach is more precise and efficient: it enables verification of (infinite-state) concurrent algorithms that prior work cannot, and for programs where prior work succeeds, our approach is faster and requires less memory.

In addition to presenting the above techniques (useful for both finite and infinite-state verification), this work represents one of the few studies on using abstract interpretation for verifying properties of infinite-state concurrent programs running on relaxed memory models and what's more, our approach requires no user annotations.

Main contributions The main contributions of this paper are:

- A new abstraction for the store buffers of the memory model that eliminates expensive shifting of buffer contents. This abstraction reduces the workload on subsequent program analyzers and improves their scalability and precision.
- A source-to-source transformation that realizes the new abstraction (and the memory model effects), producing a program that can be soundly analyzed with verifiers for sequential consistency. The translation also leverages knowledge of the underlying abstract domain in order to encode the size of the store buffers in a way which reduces the overall loss of analysis precision.
- A complete implementation of the approach integrated with CONCURINTERPROC [12], a tool based on abstract interpretation [8] with numerical abstract domains that can analyze infinite-state concurrent programs under sequential consistency.
- A thorough empirical evaluation on a range of challenging concurrent algorithms. Experimental results indicate that our technique is superior in both precision and efficiency to prior work and enables verification, for the first time, of several concurrent algorithms running on Intel's x86 TSO and PSO memory models.

2 Overview

In this section we illustrate our approach on a running example. The goal of this section is to give some intuition about and informal understanding of the work. Full formal details are provided in later sections.

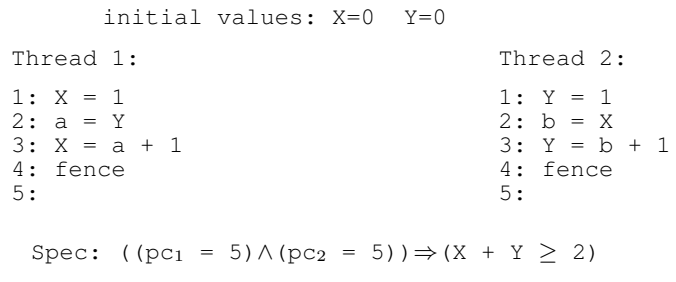

Fig. 1: Example program

To understand our approach, consider the concurrent program shown in Fig. 1. It consists of two threads that share the integer variables `X` and `Y` (variables `a` and `b` are local to each thread). The figure also shows an assertion which holds once both threads have completed their execution, namely that `X + Y` ≥ 2. Our objective is to verify that the program satisfies this assertion under relaxed memory models such as Intel's x86 TSO and PSO.

2.1 Relaxed Behaviors

In the example in Fig. 1, Thread 1 can execute the statements at labels `1` and `2` in the opposite order. Similarly, Thread 2 can execute the statements at labels `1` and `2` in the opposite order due to the nature of relaxed memory models such as TSO. Relaxed models such as TSO allow program statements to be executed out of order, resulting in behaviors not possible under sequential consistency. Under TSO, a store and a load (by the same thread) accessing different memory locations are allowed to be reordered. Therefore after both threads execute the statements at the labels `1` and `2`, one can end up in a state where the state is `X = Y = 0`. This state is impossible to obtain under sequential consistency (SC), yet is allowed under TSO. Weaker models such as PSO allow not only the reordering of store and load instructions but even the reordering of two stores (if they access different memory locations). In general, such reorderings are possible because the processor maintains store buffers per each thread, and delays expensive writes to shared memory. For instance, in Intel's x86 TSO, every thread updates a FIFO store buffer where the thread enqueues its shared memory writes and the memory sub-system dequeues these buffered writes (in the order of least recent write first) non-deterministically and updates shared memory.

2.2 SC equivalence vs. Flexible safety specifications

When considering the problem of verifying programs running on relaxed models, there are two general choices for how we select the safety property to be verified, each influencing the design of the analysis abstraction. One direction is to develop analyzers that try to prove and (if need to) enforce that the relaxed program produces results equivalent to the sequentially consistent program (and, if not equivalent, to insert fences that

make it so). This line of work was pioneered by Shasha and Snir [22], with various works later improving on the precision of the analysis and fence inference [23, 2].

Another direction, and the one pursued in this paper, is to develop analyzers which can enforce arbitrary safety properties, not only equivalence. This is advantageous for two reasons:

(i) the relaxed program might produce behaviors which are valid yet do not exist under SC, and enforcing equivalence leads to generation of redundant fences. To illustrate this point, consider the program in Fig. 1. As mentioned before, the state `X = Y = 1` is reachable under TSO at the end of the program. This state is impossible to reach under SC. If we aim to achieve SC equivalence, additional `fence` statements should be inserted in the program to prevent re-orderings that lead to this state. If we focus on ensuring the safety specification, only the current fences at labels `4` in the two threads are sufficient for verification; and

(ii) even if equivalence is the right specification, it may be difficult to produce an analysis that can prove equivalence; writing a more program specific, flexible safety property (which enforces the same constraints) may be easier to verify. We illustrate this point in Section 6: we show that [2] produces redundant fences, which our analysis avoids.

2.3 Our Approach

We now discuss the flow taken in this work. For ease of presentation, we directly present the source-to-source transformation with the abstraction embedded into that transformation.

Step 1: Buffer analysis A preliminary step of our approach is a buffer-size analysis of the input program (recall that a buffer exists in each thread). This analysis outputs an over-approximation of the size of the write buffer at each point in the program. For our running example, the analysis determines that at line `1` (of both threads), the maximum write buffer size is 1, at line `3` the maximum buffer size is 2, and at line `5`, the maximum buffer size is 0 (due to `fence`).

Step 2: Abstraction and source-to-source transformation A key step of our approach is an abstraction that eliminates buffer shifting and a source-to-source transformation realizing that abstraction (we focus on presenting both together). Here, the write buffer of each thread is directly encoded into the source code of the target program. The transformation (with abstraction) proceeds by processing the original program in a statement by statement manner. In Fig. 2, we show the result of applying our transformation for TSO on the statements of Thread 1. We next informally discuss this procedure.

To encode the store buffers used by the relaxed memory model, we introduce two kinds of variables. An example of the first kind is $\texttt{X}_1\texttt{t}_1$, which captures the value of the first write to shared variable `X` found in the buffer of thread $\texttt{t}_1$. An example of the second kind is the boolean variable $\texttt{flagX}_1\texttt{t}_1$, which denotes whether or not the first element of the write buffer of $\texttt{t}_1$ stores a write to shared variable `X` (as in general the first write found in the buffer of thread $\texttt{t}_1$ could be to some other shared variable).

Returning to our example, since the buffer is initially empty, the statement `X = 1` is translated to two updates. First, the new variable X_1t_1 is updated and set to `1`, and second, the boolean variable $flagX_1t_1$ is set to `true`.

However, simply updating the two newly generated variables is not enough because under TSO (and PSO), the memory sub-system can trigger a non-deterministic flush of a thread's store buffer at any point (the flush operation dequeues the least recent write in the buffer and updates shared memory with that write). To capture this behavior, we add a special flush code fragment after every program statement. Therefore, in our example, a flush is added after the statements at labels `1`, `2` and `3`. The flush code fragments following the statements at labels `1` and `2` are identical. The loop captures the non-deterministic effects of the flush semantics: either the flush takes place and the write stored in X_1t_1 is flushed to shared memory (and if so, the boolean variable $flagX_1t_1$ is reset to `false`), or the program continues with no changes.

Statement `a = Y` is translated without change as the buffer size analysis determines that `Y` is never written to by Thread 1 and hence the value is always read from shared memory (as opposed to the buffer).

Original statement:	Transformed statement:
`X = 1`	X_1t_1 = 1 $flagX_1t_1$ = **true**

`flush`	**while** random **do**
	  if $flagX_1t_1$ **then**
	  $X = X_1t_1$
	  $flagX_1t_1$ = **false**

`a = Y`	`a = Y`

`flush`	**while** random **do**
	  if $flagX_1t_1$ **then**
	  $X = X_1t_1$
	  $flagX_1t_1$ = **false**

`X = a + 1`	**if** $flagX_1t_1$ **then**
	  X_2t_1 = a + 1
	  $flagX_2t_1$ = **true**
	else
	  X_1t_1 = a + 1
	  $flagX_1t_1$ = **true**

`flush`	**while** random **do**
	  if $flagX_1t_1$ **then**
	  $X = X_1t_1$
	  $flagX_1t_1$ = **false**
	  else if $flagX_2t_1$ **then**
	  $X = X_2t_1$
	  $flagX_2t_1$ = **false**

`fence`	assume(¬ $flagX_1t_1$ ∧ ¬ $flagX_2t_1$)

Fig. 2: The result of applying our transformation for TSO on Thread 1 from Fig. 1. The `flush` statements are not part of the program but need to be captured by the translation (and are inserted after every program statement).

Next, statement `X = a + 1` is translated. The generated code fragment first tests if $flagX_1t_1$ is set to `true`. This answers the question of whether the first position in the buffer is already taken. We need this test as it is statically unknown whether a non-deterministic flush has fired. Depending on the result of the test, we now know where to write the value `a + 1`. If the first position of the write buffer is occupied, `a + 1` is stored to the second element of the write buffer and the appropriate flag is set (i.e., $flagX_2t_1$ is set to `true`). Otherwise, we store the value `a + 1` to the first position in the buffer and set the appropriate flag.

We next generate the flush code fragment after the statement at label `3`. This flush code is slightly different than the previous two flush fragment because at this point in the translation, the buffer-size analysis indicates that the maximum possible buffer size

is 2. Therefore, we need to dynamically check what the actual size of the buffer is and flush the appropriate entry. This can either be the variable X_1t_1 or the variable X_2t_1. Naturally, once the write to shared memory is completed, we set the corresponding auxiliary boolean variable accordingly: $flagX_1t_1$ or $flagX_2t_1$.

A key point is that we *do not* shift the store buffer contents on `flush` as a direct encoding of the memory model would do (and as previous approaches do ; see [10], [18]. Doing less work on a `flush` leads to more precise analysis and greater efficiency than prior work.

Finally, the fence statement at label `4` ensures that all writes before the fence are flushed to shared memory. An assume statement on both boolean variables captures this requirement.

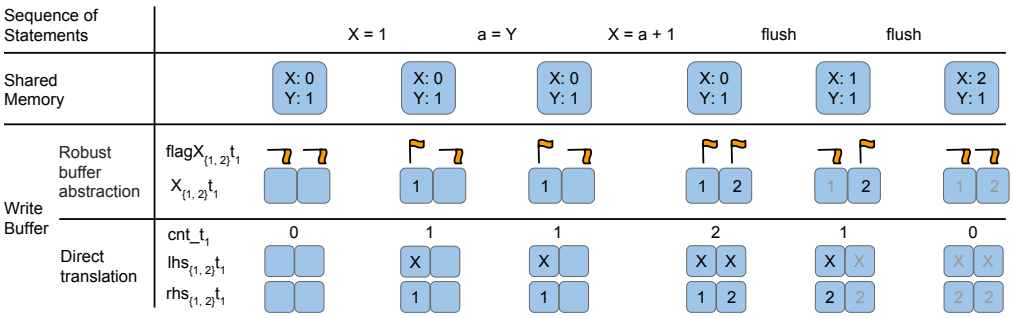

Fig. 3: The effect of a program trace on shared state and the state used by the two translations. The figure shows only statements of Thread 1 as well as flushes affecting Thread 1's write buffer.

An example trace In Fig. 3 we illustrate how a particular program trace updates the shared memory and the newly generated variables. The first line of that figure shows the sequence of statements in the trace. The second line shows the shared memory state (before and after each statement is executed). The third line (titled "robust buffer abstraction") shows the values of the newly generated variables. Here, the first square box corresponds to X_1t_1 and the second square box corresponds to X_2t_1. Similarly, the first flag corresponds to $flagX_1t_1$ and the second flag to $flagX_2t_1$. If a flag is raised, it means the variable is set to `true`; otherwise it is set to `false`. For now we can ignore the fourth line (this is a previous transformation used by [10] and [18] and is discussed later in the paper in Section 4). The trace we show and discuss is:

(i) initially, $flagX_1t_1$ and $flagX_2t_1$ are set to `false` and shared variables `X` and `Y` contain 0;
(ii) thread 2 executes `Y = 1` and a flush updates `Y` in shared memory (the trace in Fig. 3 starts after this step);
(iii) thread 1 executes `X = 1` resulting in $flagX_1t_1$ being set to `true` and X_1t_1 containing the value 1;
(iv) thread 1 reads `a = Y`, obtaining the value 1 (Fig. 3 omits local variable a, so no changes are shown);

(v) thread 1 executes `X = a + 1` resulting in $\texttt{flagX}_2\texttt{t}_1$ being set to `true` and $\texttt{X}_2\texttt{t}_1$ containing the value 2 at which point we have two writes in the store buffer of Thread 1;
(vi) a flush of Thread 1's buffer results in $\texttt{X}_1\texttt{t}_1$'s value being written to shared memory setting `X` to 1 and $\texttt{flagX}_1\texttt{t}_1$ is set to `false` to mark that the flush completed;
(vii) a flush of Thread 1's buffer results in `X` being set to 2 in shared memory and in setting $\texttt{flagX}_2\texttt{t}_1$ to `false`;

Step 3: Program Analysis Once the translated (and potentially infinite-state) concurrent program is obtained, the final step is to analyze it and attempt to prove the property of interest. Any analysis can be used; in this work we chose logico-numerical abstract domains for the following reasons: (i) there are readily available tools that implement these domains (e.g., we use CONCURINTERPROC, which implements convex numerical domains combined with boolean values), allowing us to focus on the novel parts of the work, and (ii) our benchmarks manipulate numerical variables and the properties we prove depend only on such numerical manipulations. We do note, however, that our abstraction can be useful in any setting, not just that of abstract interpretation.

The resulting analysis outputs invariants for each pair of thread locations. For instance, at labels `5`, when both threads have completed, a fragment of the resulting invariant produced by the analysis is:

$$\neg\texttt{flagX}_1\texttt{t}_1 \wedge \neg\texttt{flagX}_2\texttt{t}_1 \wedge \texttt{X} \geq \texttt{X}_1\texttt{t}_1 \wedge \texttt{X}_1\texttt{t}_1 \geq 1 \wedge ...$$

This invariant contains both a boolean part, consisting of concrete values for the auxiliary variables $\texttt{flagX}_1\texttt{t}_1$ and $\texttt{flagX}_2\texttt{t}_1$, and a numerical part in the polyhedra numerical domain: $\texttt{X} \geq \texttt{X}_1\texttt{t}_1$ and $\texttt{X}_1\texttt{t}_1 \geq 1$.

Both auxiliary boolean variables are `false`, which corresponds to an empty write buffer for Thread 1. From the numerical inequalities, we conclude that $\texttt{X} \geq 1$. Similar constraints are obtained for the variables in Thread 2, allowing us to conclude that $\texttt{Y} \geq 1$. Thus, we can conclude that the specification $\texttt{X + Y} \geq 2$ holds when both threads terminate.

We note that for our running example, direct handling of write buffer contents as used in [18] fails to verify the specification, even though the program satisfies it. This is because a direct, shift-based handling causes precision loss during the analysis. In the next section, we formally present our abstraction and transformation, discuss how it compares to prior work, and show why it leads to more scalable and precise analysis.

3 Background

In this section we provide a brief review of previous direct encoding techniques as well as terms that will be useful for our new abstraction in Section 4.

3.1 Direct source-to-source encoding

Let $Prog$ be the set of all programs, Rmm be the set of relaxed memory models (in this paper $Rmm = \{x86\ TSO, PSO\}$), and $\mathbb{N}$ the natural numbers. The translation

mechanism can be seen as a function with the signature: $T : (Prog \times Rmm \times \mathbb{N}) \rightarrow Prog$ where $P \in Prog$ is an input program, $M \in Rmm$ is a relaxed memory model, and $b \in \mathbb{N}$ is a bound on the buffer size.

The meaning of buffer size bound b Key elements of the x86 TSO memory model (and the PSO memory model) are the store buffers found between each thread and shared memory. Given buffer size bound b, the output of the translation is a new program $P_M \in Prog$ where $P_M = T(P, M, b)$.

By construction, the behavior of P_M under sequential consistency semantics captures the behavior of P under the relaxed model M, with the exception of potentially overflowing the store buffer. That is, if during the execution of P_M an attempt is made to store more than b elements to the buffer, then the program P_M aborts.

If we manage to verify that P_M satisfies the specification (without aborting), we can guarantee that P satisfies the specification under the memory model M. If the program P_M aborts, we may have to refine our model and retry verification with a larger buffer size.

It is generally impossible to statically determine the maximal store buffer size reachable during a program execution. However, in practice, static analysis can over-approximate the maximal possible store buffer size. We distinguish two cases: (i) the over-approximated value is finite. In this case, the buffer size over-approximation is useful in optimizing the transformation procedure, and (ii) the over-approximated value is unbounded. In this case, the transformation has a fixed buffer bound defined by the user.

3.2 Direct Translation

We first discuss the intuitive, direct translation function which encodes the relaxed memory semantics into the program source code. This direct translation is used by prior works focusing on infinite-state verification [10, 18]. We denote this translation by:

$$T_D : (Prog \times Rmm \times \mathbb{N}) \rightarrow Prog.$$

In the following, we use $Local$ to denote the set of local variables (per thread) and $Shared$ the set of global shared variables. Expressions, both numerical and boolean, can refer only to local variables. Statements can read and write global variables. We use $Stmt$ to denote all statements.

The translation encodes relaxed memory store buffers using temporary variables. For each statement of $P \in Prog$ we generate a code segment that captures the relaxed behavior of that statement. We define a transformation function at statement level:

$$[\![\,]\!] \in Stmt \times Thread \times \mathbb{N} \rightarrow Stmt.$$

The direct translation introduces new variables for capturing the effect of storing values into store-buffers instead of directly into main memory. For TSO (the translation for PSO is similar), the buffer is modeled with the following local variables:

– variable identifiers: $\texttt{lhs}_1\texttt{t}$, $\texttt{lhs}_2\texttt{t}$, ..., $\texttt{lhs}_b\texttt{t}$, where b is the maximum size of the buffer. The identifier of a global variable is an integer – it stores an index of the shared variable to be written to shared memory.

– buffer content values: $\texttt{rhs}_1\texttt{t}$, $\texttt{rhs}_2\texttt{t}$, ..., $\texttt{rhs}_b\texttt{t}$ – each stores the actual value to be written to shared memory.
– buffer counter: `cnt_t` takes values in the range $[0, b]$ – it stores the size of the buffer during execution.

$[\![\text{X = r}]\!]_b^t$	$[\![\text{r = X}]\!]_b^t$	$[\![\text{flush}]\!]_b^t$	$[\![\text{fence}]\!]_b^t$

if (cnt_t=b)
  abort("overflow")
cnt_t = cnt_t+1
if (cnt_t=1)
  $\texttt{lhs}_1\texttt{t}$ = X
  $\texttt{rhs}_1\texttt{t}$ = r
...
if (cnt_t=b)
  $\texttt{lhs}_b\texttt{t}$ = X
  $\texttt{rhs}_b\texttt{t}$ = r
	if (cnt_t=n)∧
	  ($\texttt{lhs}_b\texttt{t}$=X)
	  r = $\texttt{rhs}_b\texttt{t}$
	...
	else if (cnt_t=n)∧
	  ($\texttt{lhs}_1\texttt{t}$=X)
	  r = $\texttt{rhs}_1\texttt{t}$
	else
	  r = X
		while random **do**
		  if (cnt_t>0)
		  ▷ ∀ X ∈ *Shared* :
		  if ($\texttt{lhs}_1\texttt{t}$ = X)
		  X = $\texttt{rhs}_1\texttt{t}$
		  ▷ **end**
		  if (cnt_t>1)
		  $\texttt{lhs}_1\texttt{t}$ = $\texttt{lhs}_2\texttt{t}$
		  $\texttt{rhs}_1\texttt{t}$ = $\texttt{rhs}_2\texttt{t}$
		  ...
		  if (cnt_t=b)
		  $\texttt{lhs}_{b-1}\texttt{t}$ = $\texttt{lhs}_b\texttt{t}$
		  $\texttt{rhs}_{b-1}\texttt{t}$ = $\texttt{rhs}_b\texttt{t}$
		  cnt_t = cnt_t-1
		  yield
			assume
			(cnt_t = 0)

Fig. 4: Direct TSO Translation Rules of T_D

Fig. 4 presents the rules of the direct translation. In the translation of each statement, the generated sequence of statements is atomic. An exception to that rule is the flush in which only the inside of the generated loop is atomic and context switches are allowed between the loop iterations.

Write to a global variable $[\![\textbf{X = r}]\!]_b^t$***:*** the store to a global variable X first checks whether it can exceed the buffer bound b, and if so, the program aborts. Otherwise, the counter is incremented. The rest of the logic checks the value of the counter and updates the corresponding local variables. The global variable X is not updated and only local variables are modified.

Read from a global variable $[\![\textbf{r = X}]\!]_b^t$***:*** the load from a global variable X checks the current depth of the buffer and then loads from the corresponding local variable. When the buffer is empty (i.e., `cnt_t = 0`), or the variable has no occurrences in the buffer, the load is performed directly from shared memory.

Fence statement $[\![\textbf{\textit{fence}}]\!]_b^t$***:*** the fence waits for the buffer to be empty before executing.

Flush procedure $[\![\textbf{\textit{flush}}]\!]_b^t$***:*** the flush procedure is translated into a non-deterministic loop (we use `random`). If the buffer counter is positive and the entry at position 1 in the buffer ($\texttt{lhs}_1\texttt{t}$) refers to X, then the write value at position 1 (i.e., $\texttt{rhs}_1\texttt{t}$) is stored in X. The contents of the local variables are then updated by shifting: the content of each $\texttt{X}_{j+1}\texttt{t}$ is moved to its predecessor $\texttt{X}_j\texttt{t}$ where $1 \leq j < b$. Finally, the buffer count is decremented.

To encode non-deterministic flushes of the memory sub-system, a flush procedure is added by the translation function to the output program. The role of the flush procedure is to soundly encode the possible non-deterministic flushes of the store buffer, triggered by the memory subsystem. Naively, a faithful translation of the flush action requires placing the flush code after *each* statement of the program that accesses shared memory. However, this can be optimized using a simple preliminary static analysis that finds cases where the store buffer is guaranteed to be empty (and thus no flush is needed), or guaranteed to be bounded by a fixed size (and thus the flush code can be simplified).

Trace Example Returning to Fig. 3 of Section 2, the last row of the figure (titled "Direct translation") illustrates how a given trace is processed using the direct translation. The key here is the processing of the first `flush` statement, where the contents of the store buffer are explicitly shifted. As we will see next, such explicit shifting is in fact completely avoided by our new abstraction and subsequent translation.

Shortcomings of the Direct Translation The main problem with the direct translation is that it performs operations that have a devastating effect on verification. Specifically: (i) the `flush` operation performs a shift of the array content, an operation that is *very costly* and makes it harder to track the relationships between values; (ii) the sizes of store buffers are tracked via numerical variables (i.e., `cnt_t`), the value of which may be lost under abstraction. As we show in Section 5, these shortcomings cause verification using direct translation to fail in more than 50% of our benchmarks, and to be costly for the remaining ones. In the next section, we present an abstraction and a translation which address these two shortcomings.

4 Abstraction-Guided Translation

We next present our new translation, which is based on a novel abstraction of the store buffers. We also contrast our approach with the direct encoding discussed earlier:

$$T_V : (Prog \times Rmm \times \mathbb{N}) \rightarrow Prog.$$

Our presentation focuses on the x86 TSO memory model (the details for PSO are similar). We first discuss the new abstraction, which eliminates shifting of values in the store buffers. Here, when an element is flushed from the buffer, the other elements maintain their position, significantly reducing the cost of the flush operation. This abstraction is generally applicable for any analysis. We then discuss an approach for replacing the counter variables that track the current size of the write buffers with boolean variables, which also improves precision when using abstract interpretation based analysis.

4.1 Robust Buffer Abstraction – eliminating buffer shifting

The flush procedure appears at multiple places in the resulting program and hence its operation is critical to the overall precision and scalability of the analysis. As discussed earlier, the direct translation encodes a store buffer using two *bounded* arrays per thread (i.e. `lhs` and `rhs`) and a counter. If the bound is reached during analysis, an overflow error is triggered and the analysis aborts. When this happens, the user may increase the

$[\![X = r]\!]_b^t$	$[\![r = X]\!]_b^t$	$[\![flush]\!]_b^t$	$[\![fence]\!]_b^t$

if OR(b,t)
  abort("overflow")
else
if OR(b-1,t)
  X_bt = r
  $flagX_b$t = **true**
else
if OR(b-2,t)
  ...
else
  X_1t = r
  $flagX_1$t = **true**
	if ($flagX_b$t)
	  r = X_bt
	else
	if ($flagX_{b-1}$t)
	  ...
	else
	  r = X
		while random **do**
		  yield
		  if ($flagX_1$t)
		  X = X_1t
		  $flagX_1$t = **false**
		  else
		  if ($flagY_1$t)
		  Y = Y_1t
		  $flagY_1$t = **false**
		  else
		  if ($flagX_2$t)
		  ...
			assume
			(¬**OR**(b,t) ∧
			... ∧
			¬**OR**(1,t))

Fig. 5: Abstraction-guided translation rules (i.e. T_V) for TSO.

buffer bound, transform the program using the new bound, and rerun the analysis on the newly obtained program. The flush routine in the direct translation is implemented using a non-deterministic loop. In the loop body, the first element in the store buffer (the oldest) is flushed to memory. Next, the remaining elements are *shifted* one position to the left in the buffer. An advantage of shifting is that it frees entries at the end of the arrays encoding the buffer, thus creating free space for buffering additional store operations.

Key Idea: Our observation is that we can handle the flush operation without shifting the array content, thus obtaining an abstraction (over-approximation) of the relaxed memory semantics. This approximation is sound (the proof is presented in Section 4.4) but may *introduce additional cases of overflow*. That is, if a program reaches an overflow when analyzed with our abstraction, it is possible that this overflow may not occur when using the direct, shifting encoding. However, we believe such situations are very rare in practice – in our evaluation in Section 5, no additional such overflows appeared in any of the benchmarks. We formally discuss how our abstraction is incorporated into the translation later in Section 4.3.

4.2 Replacing counters with boolean flags

Another ingredient of our approach is leveraging properties of the underlying program analysis. Unlike the general abstraction above, here we discuss an optimization suitable for abstract interpretation based analysis with numerical domains.

The direct translation keeps designated counters to track the current position in store buffers. When using numerical abstract domains such as Octagon [19] and Polyhedra [9], the exact numerical value of a variable may be abstracted away at program join points. This abstraction, desirable in most cases, has negative effects when applied to key variables such as buffer counters. We would therefore like to keep the values of buffer counters even when different values for the count reach program join points.

Towards this, we use a logico-numerical domain, which combines a numerical domain and a logical domain that tracks boolean combinations of predicates. Rather than

storing values of buffer counters as integers in the numerical part of the domain, we encode them using boolean variables in the logical part. This allows us to naturally maintain a disjunction of possible values for counters, without joining them. Using boolean variables to track buffer sizes therefore improves the precision of the analysis inside the flush procedure by differentiating cases where values of counter variables differ. This encoding can be viewed as a form of trace partitioning [20], where joins are delayed based on certain key values (in our case, the values of counter variables).

4.3 New Translation Rules

The source-to-source translation presented next incorporates both of the ideas described above. It replaces cnt_t counter variables with boolean variables. For each shared variable $\texttt{X} \in Shared$, write buffer index $i \in [1, b]$, and thread identifier $t \in Thread$, a boolean variable $\texttt{flagX}_i\texttt{t}$ is added.

If $\texttt{flagX}_i\texttt{t}$ is `true`, then there is a shared variable `X` write in the thread t write buffer, to position i.

The x86 TSO memory model has a single write buffer per thread. This translates to the invariant: for a fixed $i \in [1, b]$ and a fixed $t \in Thread$, there exists at most one shared variable `X` such that $\texttt{flagX}_i\texttt{t}$ is `true`. In other words, at each location of the TSO buffer there is at most one shared variable write. We define the function:

$$\texttt{OR(i, t)} = \vee_{X \in Shared}\ \texttt{flagX}_i\texttt{t}.$$

The function `OR(i, t)` returns `true` if there exists a write (to any shared variable) at the position i in the write buffer of thread t. The previously mentioned invariant means that at most one disjunct will be true in the formula above. Fig. 5 shows the rules of the abstraction-guided translation:

Write to a global variable $[\![\boldsymbol{X = r}]\!]_b^t$***:*** first checks if there is a write in the last element of the store buffer. If so, the analysis indicates an overflow and stops. If the store buffer is not yet full, the translation determines the highest index i in the buffer which is already occupied and places the current write at the position $i + 1$. Note that in each branch of the if-then-else statement, a boolean variable is modified. This enables the robust buffer abstraction (Sec. 4.1) and the boolean encoding of counters (Sec. 4.2).

Read from a global variable $[\![\boldsymbol{r = X}]\!]_b^t$***:*** searches in the store buffer for the most recent write to the shared variable `X` and returns that value. If there is no write to `X` in the store buffer, then the value is read from the shared memory.

Fence statement $[\![\boldsymbol{fence}]\!]_b^t$***:*** assumes that at this point the store buffer is empty – there are no pending writes.

Flush action $[\![\boldsymbol{flush}]\!]_b^t$***:*** searches for the least recent entry in the store buffer and writes it to the shared memory. As opposed to the direct encoding, the element at position 1 is not flushed *because the shifting procedure was removed*. To know which variable is the buffered write, case testing is performed.

The new translation extends naturally to a sequence of statements and to programs with n concurrent threads: $[\![P]\!]_b = [\![S]\!]_b^1 \parallel \cdots \parallel [\![S]\!]_b^n$.

4.4 Soundness of the Robust Buffer Abstraction

We next prove that the RBA abstraction incorporated in the translation T_V is sound as it over-approximates the direct translation T_D. Given a program P, memory model M, and buffer bound b, $P^D = T_D(P, M, b)$ is the program that results from applying direct translation, and $P^V = T_V(P, M, b)$ is the result of the abstraction-guided translation.

D: direct translation domain.	**V: abstraction-guided translation.**
$[b]$ = value of $cnt\_t \in \{0 \dots b\}$	$(Shared \to Bool)$ = values of $\texttt{flagX}_i\texttt{t}$
$Shared \times \mathbb{N}$ = values of $lhs_i t, rhs_i t$	$(Shared \to \mathbb{N})$ = values of $\texttt{X}_i\texttt{t}$
$B_t^D = [b] \times Seq_{\leq b}(Shared \times \mathbb{N})$	$B_t^V = Seq_{\leq b}(Shared \to (Bool \times \mathbb{N}))$

Fig. 6: Translation Domains

Fig. 6 summarizes the data structures needed to encode the write buffer of a thread t in the direct and abstraction-guided translations:

- B_t^D is the tuple containing the value of $cnt_t \in [b]$ and the sequence of pairs of values for $lhs_i \in Shared$ and $rhs_i t \in \mathbb{N}$, $i \in \{1 \dots b\}$.
- B_t^V is a sequence of elements which, for each shared variable $X \in Shared$, associate a tuple containing the boolean variable $\texttt{flagX}_i\texttt{t} \in B$ and the stored value $\texttt{X}_i\texttt{t} \in \mathbb{N}$.

Let $B^D = \{B_t^D | t \in Threads\}$ and $B^V = \{B_t^V | t \in Threads\}$ be the sets of values of all write buffers of the programs P^D and P^V.

We define the state of a translated program as the values of the shared variables, local variables, program counter, and auxiliary variables added by the translation: $\sigma = \langle Shared_\sigma, Local_\sigma, pc_\sigma, B \rangle$ or $\sigma = overflow$. B is either the direct translation buffer state B^D or the abstraction-guided translation buffer state B^V.

Definition 1 (Observable part of a state). *The observable part of a state includes: (i) the values of the shared variables, (ii) the values of the local variables, and (iii) the values and order between elements of the* non-empty section *of the buffer.*

For T_D, the observable part of the state contains the values of the shared and local variables and the values of lhs_i and rhs_i for $i \in [1 \dots cnt\_t]$. Similarly, for T_V, the observable part of the state contains the values of the shared and local variables and the values of $\texttt{X}_i\texttt{t}$ for i and t, where $\texttt{flagX}_i\texttt{t}$ is *true*.

Definition 2 (Equivalent states). *Two states σ^D and σ^V are equivalent if their observable parts correspond (the global and local variables have the same values and the buffers B^D and B^V denote the same buffer content).*

We define the transitions between two states (σ_i, σ_{i+1}) for transformed programs as the translation rule (Fig. 4 or Fig. 5) corresponding to the transition in the original program P. A trace of a program is represented as a sequence of states $\pi = \sigma_1 \dots \sigma_n$.

Theorem 1 (The RBA abstraction used in T_V is sound). *For any trace $\pi^D = \sigma_1^D \dots \sigma_s^D$ of P^D of finite length s, there exists a corresponding trace $\pi^V = \sigma_1^V \dots \sigma_s^V$ of T^V, such that for all $i \in \{1 \dots s\}$, σ_i^V and σ_i^D are equivalent or σ_i^V is overflow.*

Proof. The proof is by induction on the length of π^D.

First, we show how to build the trace π^V. Given $\pi^D = \sigma_1^D \dots \sigma_s^D$, the transition $(\sigma_i^D, \sigma_{i+1}^D)$ for $i \in [1 \dots s-1]$ is a rule in Fig. 4, corresponding to the translation of an instruction in program P. We construct π^V by applying at each step $(\sigma_i^V, \sigma_{i+1}^V)$ the corresponding rule from Fig. 5.

Next, we prove that π^V and π^D have equivalent states.

Base case: for $i = 1$, in the initial state, all write buffers are empty, the shared variables have their initial values, and the local variables are not yet declared. Thus, states σ_1^V and σ_1^D are equivalent.

Induction step: for $i > 1$, we assume that the states σ_i^V and σ_i^D are equivalent or σ_i^V is overflow. If σ_i^V is overflow, then σ_{i+1}^V is also overflow (by convention, an overflow state cannot be changed).

If σ_i^V is not overflow, then the states σ_i^V and σ_i^D are equivalent (by the induction assumption). Our construction applies the transition $(\sigma_i^D, \sigma_{i+1}^D)$ as defined by the rules in Fig. 4 and the corresponding transition $(\sigma_i^V, \sigma_{i+1}^V)$ as defined by Fig. 5. We now show that σ_{i+1}^D and σ_{i+1}^V are equivalent or σ_{i+1}^V is overflow via case splitting on the transition type:

- `store`: write to a global variable $[\![\text{X = r}]\!]_b^t$. Here, the local and shared variables remain unchanged. From the induction assumption σ_i^D and σ_i^V, buffers hold the same values in the same order. From the assumption $\sigma_i^D \neq overflow$ and from the definition of `store`, we have that buffer content in σ_i^D and σ_i^V is the same or σ_{i+1}^V will reach overflow.
- `load`: read from a global variable $[\![\text{r = X}]\!]_b^t$. Here, the buffer contents are unchanged. The shared variables are also unchanged. From the induction assumption and the definition of `load` we have that the values of r for σ_{i+1}^D and for σ_{i+1}^V are the same.
- `fence`: fence statement $[\![\text{fence}]\!]_b^t$. Here, the transition assumes that at this point the store buffers are empty for both translations. The states do not change and the assumption on σ_i^D and σ_i^V propagates to the states σ_{i+1}^D and σ_{i+1}^V.
- `flush`: flush action $[\![\text{flush}]\!]_b^t$. Here, the local variables are unchanged. From the induction assumption, the buffers of σ_i^D and σ_i^V hold the same values in the same order, i.e., the same least recent element in the buffer will be flushed to main memory for σ_{i+1}^D and σ_{i+1}^V.

This concludes the proof of Theorem 1 that T^V is an over-approximation of T^D and the RBA abstraction is sound. This also means that even if the trace π^D does not reach an overflow, the corresponding trace π^V may result in overflow.

5 Evaluation

We implemented our approach and evaluated it on a range of challenging concurrent algorithms. We then compared its performance with the direct transformation discussed earlier [18]. All our experiments ran on an Intel(R) Xeon(R) 2.13 GHz server with 250

Table 1: Verification results comparing our new transformation with prior work [18]

Program	Model	Number fences	**Abstraction-guided translation** Time (sec)	Memory (MB)	**Direct translation** [18] Time (sec)	Memory (MB)
Abp	TSO	0	5	189	14	352
	PSO	0	6	167	12	222
Bakery	TSO	4	1148	4749	-	-
	PSO	4	3429	10951	-	-
Concloop	TSO	2	8	547	18	891
	PSO	2	6	504	23	783
Dekker	TSO	6	227	2233	-	-
	PSO	4	121	1580	-	-
Kessel	TSO	4	14	357	15	424
	PSO	4	6	198	80	628
Loop2 TLM	TSO	2	66	2234	-	-
	PSO	2	36	1650	-	-
Peterson	TSO	2	89	1549	-	-
	PSO	4	20	901	331	2280
Pgsql	TSO	3	282	1727	-	-
	PSO	1	55	758	-	-
Queue	TSO	1	1	101	1	115
	PSO	1	1	108	1	106
Sober	TSO	2	30	1784	-	-
	PSO	3	148	263	215	3499
Szymanski	TSO	3	1066	3781	-	-
	PSO	4	507	2076	-	-
Chase-Lev WSQ	TSO	2	17	550	-	-
	PSO	4	9	520	10	528
THE WSQ	TSO	4	125	1646	-	-
	PSO	4	391	2338	-	-

GB RAM. To perform the analysis, we used CONCURINTERPROC [12], a tool based on the APRON library [13], which supports various numerical abstract domains. We relied on the Z3 [11] SMT solver to check that the inferred invariants imply the specification.

The verification procedure has three steps:

(i) Applying the transformation on program P, obtaining a new program P_M.
(ii) Running CONCURINTERPROC on that transformed program.
(iii) Using Z3 to check whether the inferred invariants satisfy the specification.

The above procedure is repeated until it is no longer possible to further reduce the number of fences in the algorithm. We evaluated our approach on 13 concurrent algorithms, out of which 5 are infinite-state. The safety specifications are either mutual exclusion or reachability invariants involving labels of different threads.

Our main goal was to study the Abstraction-guided translation precision and efficiency gains (i.e. memory consumption, speed) compared to the direct translation [18], while using the same analysis tool (in this case CONCURINTERPROC) to verify the output programs. Where applicable, we also discuss how our work compares to other works that are state of the art(here and in Section 6). Table 1 summarizes our experimental results for both the x86 TSO and PSO memory models.

The minimal number of fences necessary to verify each algorithm are shown in column 3 of Table 1. The time and memory resources used by the analysis are shown for both the new transformation (in columns 4 and 5 of the table) and the previous transformation (in columns 6 and 7). We observe two trends:

- For Bakery, Dekker, Loop2-TLM, Pgsql, Szymanski and THE WSQ, the new transformation verifies the program with strictly fewer fences than the direct translation. The dash indicates that verification failed (out of memory or timeout) for those placements (or their subsets) using the direct translation.
- For the rest of the benchmarks, the direct translation was successful in verifying the same fence placement as our new translation. But in all those cases the time and memory consumption were better using the new translation, and in some instances (e.g., Sober) memory consumption was reduced by 10x.

Comparison to other work Recent work [2] infers fences such that the program under the relaxed model is equivalent to SC – recall that we discussed such approaches as one of two general approaches in Section 3. Although scalable, the authors' abstraction tends to lead to significant precision loss, thus inserting redundant fences. For instance, in Lamport's Bakery under TSO, their abstraction inserts 8 fences, compared to 4 fences inserted by our analysis. This precision loss is observed also for other mutual exclusion algorithms such as Peterson under TSO (3 vs. 2 fences) and Szymansky under TSO (8 vs. 3 fences).

Another line of work [3] also produces an SC program from the original program and the relaxed memory model semantics. The work uses testing to find bugs in many litmus tests and algorithms (e.g., Bakery, Peterson, Dekker, Szymanski), but does not actually perform verification on any of them. Nor does it address the problem of how the proposed translation would affect infinite-state verification. For instance, when we tried [3] even on a few small examples, the resulting SC program used many more auxiliary boolean variables than our translation (e.g., 40 vs. 8). Note that even a small increase in the number of boolean variables quickly leads to state explosion (more disjunctions) in the program analysis. This is also confirmed by our experiments with CONCURINTERPROC, where, for instance, any program with more than 40 boolean variables could not be verified due to state explosion.

Summary of Results In summary, for each program, our new transformation enables verification with a lower or equal number of fences compared to the direct translation. The new transformation also leads to a more efficient (in space and time) subsequent analysis of the resulting program. Based on our experimental results, we believe that our new abstraction-gudied transformation is a key building block in automating verification of both finite and infinite-state concurrent programs on relaxed memory models.

6 Related Work

We next discuss some additional work most closely related to ours. Over the last few years there has been significant interest in ensuring correctness (via synthesis and verification) of concurrent programs running on relaxed memory models. Most of the research has so far considered only finite-state programs [5–7, 14, 15, 17]. (Some of these

papers however, do handle specific kinds of infinite-state problems, such as unbounded store buffers, but where all shared variables range over finite domains). Some recent works also handle infinite-state programs [1, 2, 10, 18, 21].

One approach to handling relaxed memory models is to encode the effect of the model directly into the program and then analyze the resulting program using tools that work for sequential consistency (e.g., [3, 4, 10]). We follow the same general idea. The main contribution of our work is a new abstraction and a transformation which improves the precision and efficiency of the resulting program analysis. For instance, as we showed in the paper, using the direct encoding as in [18] will result in significant loss of precision and efficiency (i.e., failure to verify correct programs). Abdulla et al. [1] explore online predicate abstraction for handling infinite-state verification while Dan et al. [10] also explore predicate abstraction but this time based on offline analysis of boolean programs. Technically, these works are quite different from ours since: (i) they both use direct encoding and also (ii) they both use predicate abstraction which, even with abstraction refinement, tends to require manually supplied predicates. In contrast, we provide a new robust abstraction of the store buffers and explore the application of numerical abstract domains that do not require manual annotations. We also provide a more comprehensive experimental study than either of aforementioned works (we consider x86 TSO, as well as the more relaxed PSO model and a range of challenging concurrent algorithms). For the common benchmarks, [1] and our approach achieve comparable results. A possible limitation of this work is locked writes, meaning that fences are generated immediately following a write to shared memory. Our tool is more flexible since fences can be placed at any label. We again note that the robust buffer abstraction (RBA) proposed in this work can be immediately useful with predicate abstraction as well. In the work of [21], arbitrary safety properties are not taken into account. This work supports two fence removal optimizations (for TSO), which are not enough to eliminate redundant fences. We applied their optimizations on a few of our benchmarks and, unfortunately, it failed to remove redundant fences (e.g., in the Chase WSQ algorithm).

7 Conclusion

We proposed a new approach for verifying concurrent programs on relaxed memory models. Our approach consists of a robust abstraction of the store buffers, an encoding of the store buffer sizes that leverages the underlying abstract domains, and a source to source translation that encodes relaxed memory model semantics into the target program, thereby enabling the use of existing verification tools for sequential consistency.

We implemented our approach and evaluated it on a set of finite and infinite-state concurrent algorithms using an existing state-of-the art abstract interpretation engine. Our experimental results demonstrate that the overall system is superior to prior work in terms of precision and performance, enabling verification of concurrent algorithms on both x86 TSO and PSO memory models not possible before.

References

1. ABDULLA, P. A., ATIG, M. F., CHEN, Y.-F., LEONARDSSON, C., AND REZINE, A. Automatic fence insertion in integer programs via predicate abstraction. SAS'12.
2. ALGLAVE, J., KROENING, D., NIMAL, V., AND POETZL, D. Don't sit on the fence - a static analysis approach to automatic fence insertion. In *CAV'14*.
3. ALGLAVE, J., KROENING, D., NIMAL, V., AND TAUTSCHNIG, M. Software verification for weak memory via program transformation. ESOP'13.
4. ATIG, M. F., BOUAJJANI, A., AND PARLATO, G. Getting rid of store-buffers in tso analysis. In *CAV'11*.
5. BOUAJJANI, A., DEREVENETC, E., AND MEYER, R. Checking and enforcing robustness against tso. In *ESOP'13* (2013).
6. BURCKHARDT, S., AND MUSUVATHI, M. Effective program verification for relaxed memory models. In *CAV '08* (2008).
7. BURNIM, J., SEN, K., AND STERGIOU, C. Testing concurrent programs on relaxed memory models. In *ISSTA '11* (2011).
8. COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL'77*.
9. COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among variables of a program. In *POPL'78* (1978).
10. DAN, A. M., MESHMAN, Y., VECHEV, M. T., AND YAHAV, E. Predicate abstraction for relaxed memory models. In *SAS'13* (2013).
11. DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *TACAS'08* (2008).
12. JEANNET, B. Relational interprocedural verification of concurrent programs. *Software and System Modeling 12*, 2 (2013), 285–306.
13. JEANNET, B., AND MINÉ, A. Apron: A library of numerical abstract domains for static analysis. In *CAV* (2009), A. Bouajjani and O. Maler, Eds., vol. 5643, pp. 661–667.
14. KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Automatic inference of memory fences. In *FMCAD '10* (2010).
15. KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Partial-coherence abstractions for relaxed memory models. PLDI '11.
16. LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*, 9 (Sept. 1979), 690–691.
17. LINDEN, A., AND WOLPER, P. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN* (2010), pp. 212–226.
18. MESHMAN, Y., DAN, A. M., VECHEV, M. T., AND YAHAV, E. Synthesis of memory fences via refinement propagation. In *SAS'14* (2014).
19. MINÉ, A. The octagon abstract domain. *Higher Order Symbol. Comput. 19*, 1 (2006), 31–100.
20. RIVAL, X., AND MAUBORGNE, L. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst. 29*, 5 (2007), 26.
21. SEVCÍK, J., VAFEIADIS, V., NARDELLI, F. Z., JAGANNATHAN, S., AND SEWELL, P. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM 60*, 3 (2013), 22.
22. SHASHA, D., AND SNIR, M. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst. 10*, 2 (Apr. 1988), 282–312.
23. SURA, Z., FANG, X., WONG, C.-L., MIDKIFF, S. P., LEE, J., AND PADUA, D. Compiler techniques for high performance sequentially consistent java programs. In *PPoPP '05*.