Predicate Abstraction for Relaxed Memory Models

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

[1] ETH Zurich
{andrei.dan, martin.vechev}@inf.ethz.ch
[2] Technion
{yurime, yahave}@cs.technion.ac.il

Abstract. We present a novel approach for predicate abstraction of programs running on relaxed memory models. Our approach consists of two steps.
First, we reduce the problem of verifying a program P running on a memory model M to the problem of verifying a program P_M that captures an abstraction of M as part of the program.
Second, we present a new technique for discovering predicates that enable verification of P_M. The core idea is to extrapolate from the predicates used to verify P under sequential consistency. A key new concept is that of cube extrapolation: it successfully avoids exponential state explosion when abstracting P_M.
We implemented our approach for the x86 TSO and PSO memory models and showed that predicates discovered via extrapolation are powerful enough to verify several challenging concurrent programs. This is the first time some of these programs have been verified for a model as relaxed as PSO.

1 Introduction

One approach for efficiently utilizing multi-core architectures, used by major CPU designs (e.g., [26, 27, 19]), is to define architectural *relaxed (weak) memory models (RMMs)* [13]. Some of those relaxations can be modeled using one or more per-processor FIFO buffers, where a store operation adds values to the buffer and a flush operation propagates the stored value to main memory. Programs running under those models exhibit unique caveats and verifying their correctness is challenging.

The Problem Given a program P, a specification S and a memory model M, we would like to answer whether P satisfies S under M, denoted as $P \models_M S$.

Unfortunately, even for finite-state programs, automatic verification under relaxed memory models is a hard problem. The problem is either undecidable or has a non-primitive recursive complexity for stronger models such as x86 TSO and PSO (see [4] for details). It is therefore natural to explore the use of abstraction for verification of such programs.

Predicate abstraction [15] is a widely used approach for abstract interpretation [9]. Since predicate abstraction has been successfully applied to verify a wide range of sequential and concurrent programs (e.g., [6, 12, 14]), we are interested in the question:

how to apply predicate abstraction to programs running on relaxed models?

Given a program P and a vocabulary (set of predicates) $V = \{p_1, \ldots, p_n\}$ with corresponding boolean variables $\hat{V} = \{b_1, \ldots, b_n\}$, standard predicate abstraction (e.g. [15, 6]) constructs a boolean program $\mathcal{BP}(P, V)$ that conservatively represents the behaviors of P using only boolean variables from $\hat{V}$. When considering predicate abstraction in the context of relaxed memory models, two key challenges need to be addressed: (i) *soundness*: the boolean program must faithfully abstract the behaviors of P running on model M; (ii) *predicate discovery*: there should be a mechanism for automatically discovering predicates that enable successful verification of P running on memory model M.

Soundness Under sequential consistency (SC), predicate abstraction is sound and we know that $\mathcal{BP}(P, V) \models_{SC} S$ implies $P \models_{SC} S$. Unfortunately, we observed this does not hold for relaxed memory models (see Section A.3).

Intuitively, the problem is as follows: under sequential consistency, a shared variable has only one value — the value stored in main memory. Predicates used for predicate abstraction can therefore refer to that shared value and relate it to the values of other shared variables or thread-local variables. In contrast, for a program running on a relaxed model, threads may observe *different values for the same shared variable* as they have their own local buffered copies. This means that in a relaxed model, one cannot directly apply classic predicate abstraction, as the variables used in predicates are assumed to refer to a single value at a time.

Predicate Discovery A key challenge with predicate abstraction is to discover a set of predicates that enable verification. Following classic abstraction refinement, one would start with a program that is to be verified on a particular relaxed model together with an initial set of predicates. Then, proceed to iteratively apply refinement until we find a set of predicates under which the program verifies (or the process times out).

We take a different approach to predicate discovery for programs running on RMMs. In our approach, we first obtain the predicates that enable verification of the program on sequential consistency (SC). Then, we automatically *extrapolate* from these SC predicates to produce a new set of predicates that can be used as a basis for verification on the relaxed model.

Our Approach Given a program P, a specification S and a memory model M, our approach consists of the following steps:

1. *verify under SC*: find a set of predicates V, sufficient to verify P under sequential consistency, i.e., a set V such that $\mathcal{BP}(P, V) \models_{SC} S$.
2. *reduce to SC*: automatically construct 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 store buffers used in M.
3. *discover new predicates*: automatically compute a new set of predicates V_M that are used for predicate abstraction of P_M. This is a challenging step and the key idea is to leverage the verification of P under SC. We present two approaches: predicate extrapolation which discovers new predicates based on the predicates in V and cube extrapolation which discovers new predicates based on both V and $\mathcal{BP}(P, V)$.

4. *construct a new boolean program*: given the new program P_M and the new predicates V_M, automatically construct a boolean program $\mathcal{BP}(P_M, V_M)$ such that $\mathcal{BP}(P_M, V_M) \models_{SC} S$ ensures that $P_M \models_{SC} S$, which in turn guarantees that $P \models_M S$. Here, cube extrapolation enables us to build $\mathcal{BP}(P_M, V_M)$ *without* suffering from the usual problem of exponential search.
5. *check*: whether $\mathcal{BP}(P_M, V_M) \models_{SC} S$.

Main Contributions

- We provide a novel approach for predicate abstraction of programs running on relaxed memory models, extrapolating from the predicate abstraction proof of the same program for sequential consistency.
- One of our insights is that the predicates used to verify P under SC can be automatically *extrapolated* to discover new predicates for verification of the program with M encoded in it, P_M. We present two approaches for discovering new predicates called predicate extrapolation and cube extrapolation.
- We instantiated our approach for the x86 TSO and PSO memory models. We implemented our approach and applied it to verify several challenging concurrent algorithms (both finite and infinite-state) under these models. We show that extrapolation is powerful enough to verify these algorithms and in particular, cube extrapolation enables verification of Lamport's Bakery algorithm, which otherwise (without cube extrapolation) times out when building the boolean program.

2 Overview

In this section, we give an informal overview of our approach using simple examples.

2.1 Motivating Example

Fig. 1 shows an implementation of an infinite state alternating bit protocol (ABP) with two concurrent threads. We use capitalized variable names to denote global shared variables, and variable names in lowercase to denote local variables. In this program, global shared variables `Msg` and `Ack` have an initial value 0. We use this algorithm as our illustrative example, additional examples are discussed in Section 6.

Specification When executing on a sequentially consistent memory model, this program satisfies the invariant:

$$((lRCnt = lSCnt) \vee ((lRCnt + 1) = lSCnt))$$

Here, the local variable $lRCnt$ is the local counter for the receiver thread containing the number of received messages. Similarly, local variable $lSCnt$ is the local counter for the sender thread containing the number of sent messages.

Predicate Abstraction under Sequential Consistency A traditional approach to predicate abstraction is shown in Fig. 2(a). To verify that ABP satisfies its specification under SC, we instantiate predicate abstraction with the following predicates:

$$(Msg = 0), (Ack = 0), (lSSt = 0), (lAck = 0)$$
$$(lMsg = 0), (lRSt = 0), (lRCnt = lSCnt), ((lRCnt + 1) = lSCnt)$$

initially: `Msg = Ack = 0`

Sender (thread 0):

```
1   lAck = Ack;
2   if ((lAck = 0 & lSSt = 0)
    | (lAck != 0 & lSSt != 0))
3
4           if (lSSt != 0) lSSt = 0;
5           else  lSSt = 1;
6           lSCnt++;
7           Msg = lSSt;
8   goto 1;
```

Receiver (thread 1):

```
1   lMsg = Msg;
2   if ((lMsg = 0 & lRSt != 0)
    |   (lMsg = 0 & lRSt != 0))
3
4           lRSt = lMsg;
5           lRCnt++;
6           Ack = lRSt;
7
8   goto 1;
```

Fig. 1. An alternating bit protocol example with two threads.

The result of predicate abstraction using these predicates is a concurrent boolean program that conservatively represents all behaviors of the original ABP program. In Fig. 2(a), this boolean program is denoted as the oval named Boolean Program B.

In the worst case, the construction of the concurrent boolean program, following standard predicate abstraction techniques (e.g., [15, 6, 14]) involves an exponential number of calls to an underlying theorem prover. A critical part of the construction of the boolean program is searching the "cubes" — conjunctions of predicates — that imply a certain condition. This search is exponential in the number of predicates.

An Informal Overview of PSO In the partial-store-order (PSO) memory model, each thread maintains a private buffer (a sequence) for each global variable. When the thread writes to a global variable, the value is enqueued into the buffer for that variable. Nondeterministically, the values can be dequeued from the buffer and written to main memory. When the thread reads from a global variable, it first checks if the buffer is empty and if so, it reads as usual from main memory. Otherwise, it reads the last value written in the buffer. The model is further equipped with a special *fence* instruction which can be executed by a thread to empty all buffers for that thread and write the most recent value in each buffer to the corresponding shared location. In our example, thread 0 maintains one buffer, the buffer for global variable Msg and thread 1 also maintains one buffer, the buffer for global variable Ack. For instance, the store `Msg = lSST` leads to the value of `lSST` being enqueued into the private buffer for thread 0. This value can be flushed to main memory at any point in the execution, non-deterministically.

An Informal Overview of TSO In the total-store-order (TSO) memory model, each thread maintains a single private buffer for all global variables. Similarly to PSO, values stored to global variables are first written to the buffer, and then are non-deterministically flushed to main memory. A *fence* instruction empties the thread's private buffer.

The challenges we address are: (i) how to verify programs such as ABP under relaxed memory models such as x86 TSO and PSO, and (ii) how to deal with the exponential complexity of standard predicate abstraction in our setting of RMM.

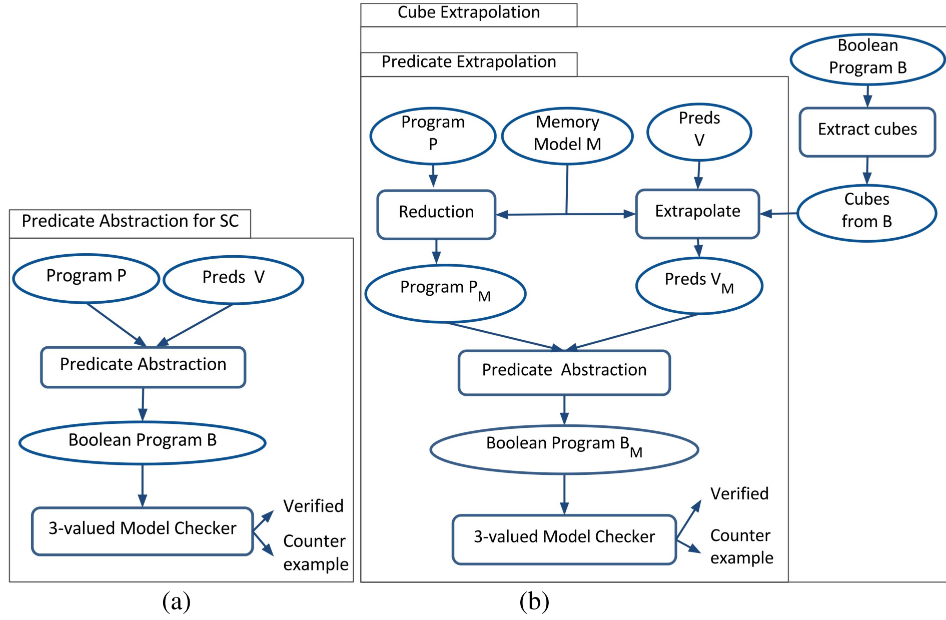

Fig. 2. Predicate abstraction: (a) classic algorithm, (b) with predicate extrapolation and cube extrapolation. Here, a rectangle shape represents an algorithm, while an oval shape represents input-output information (data).

2.2 Standard (Naive) Predicate Abstraction under RMM is Unsound

We now consider the following scheme for predicate abstraction: (i) construct a boolean program directly from the program of Fig. 10 using standard predicate abstraction; (ii) execute the boolean program using PSO semantics. This scheme is simple, but unfortunately, it is unsound. We now show that following this scheme, we can produce a boolean program that successfully verifies our assertion. This is unsound because we know the existence of an assertion violation, namely the one shown in Fig. 11(a).

```
                  initial: X=Y=0
Thread 0:                          Thread 1:
1  X = Y+1                         1  Y = X+1
2  fence(X)                        2  fence(Y)

      assert(X≠Y)
```

Fig. 3. unsoundness example.

To capture the assertion breach, we need to keep track of the relationship between X and Y. Consider the predicates:

$$P_1 : X = Y, P_2 : X = 1, P_3 : Y = 1, P_4 : X = 0, P_5 : Y = 0$$

Each P_i has been assigned a boolean variable B_i (with a buffer for each thread) where the effect of a thread writing to X or Y will write to a local buffer of $X = Y$ of that thread, and the effect of flushing X or Y will also flush the buffer associated with $X = Y$. Unfortunately this approach is insufficient. The problem is shown in Fig. 11(b). When thread 0 updates X to 1, and thread 1 updates Y to 1, the predicate $X = Y$ will be updated to *false* in both (denoted as F) and stored in the store buffer of each thread (since neither of the two threads can see the update of the other). When the value of X is

	(a) Concrete			(b) Predicate Abstraction		
	Thread 0 (X,Y)	**Thread 1 (X,Y)**	**Global (X,Y)**	**Thread 0** (X=Y,X=1,Y=1,X=0,Y=0)	**Thread 1** (X=Y,X=1,Y=1,X=0,Y=0)	**Global** (X=Y,X=1,Y=1,X=0,Y=0)
	(0,0)	(0,0)	(0,0)	(T, F, F, T, T)	(T, F, F, T, T)	(T, F, F, T, T)
T0: X = Y+1	(1,0)	(0,0)	(0,0)	(F, T, F, F, T)	(T, F, F, T, T)	(T, F, F, T, T)
T1: Y = X+1	(1,0)	(0,1)	(0,0)	(F, T, F, F, T)	(F, F, T, T, F)	(T, F, F, T, T)
T0: flush(X)	(1,0)	(1,1)	(1,0)	(**F**, **T**, F, F, T)	(F, T, T, F, F)	(**F**, **T**, F, F, T)
T1: flush(Y)	(1,1)	(1,1)	(1,1)	(F, T, F, F, T)	(**F**, T, **T**, F, F)	(**F**, T, **T**, F,F)

Fig. 4. (a) an error trace for Fig. 10 under PSO. Thread i are values observed by thread i, Global are the values in global memory. (b) Values of predicates in a boolean program corresponding to the program of Fig. 10 under PSO semantics.

flushed from the buffer of thread 0, our setting flushes the value of the predicate $X = Y$ (F) to main memory. Similarly, when Y is flushed in thread 1, the value of $X = Y$ (F) is flushed. The main memory state after these two flushes is inconsistent, as it has $X = 1$ set to T, $Y = 1$ set to T and $X = Y$ set to F.

2.3 Predicate Abstraction for Relaxed Memory Models

In Fig. 2(b), we illustrate the ingredients and flow of our approach for solving the verification problem under relaxed memory models. The figure contains two approaches for adapting predicate abstraction to our setting called Predicate Extrapolation and Cube Extrapolation (which includes Predicate Extrapolation). Next, we discuss the steps of our approach.

Step 1: Verify P ***under SC*** The first step is to verify the program P under sequential consistency using standard predicate abstraction as outlined earlier in Fig. 2(a). Once the program is verified, we can leverage its set of predicates as well as its boolean program in the following steps.

Step 2: Construct the reduced program P_M This step is named "Reduction" in Fig. 2(b). To enable sound predicate abstraction of a program P under a relaxed memory model M, we first reduce the problem into predicate abstraction of a sequentially consistent program. We do so, by constructing a program P_M that conservatively represents the memory model M effects as part of the program.

The key idea in constructing P_M is to represent an abstraction of the store buffers of M as additional variables in P_M. Since the constructed program P_M represents (an abstraction of) the details of the underlying memory model, we can soundly apply predicate abstraction to P_M. The formal details of the reduction for x86 TSO and PSO are discussed later in the paper. Here, we give an informal description.

For PSO, it is sufficient to consider a program P_{PSO} where every global variable X in P is also associated with: (i) additional k *local* variables for each thread t: $x_{1_t}, \ldots, x_{k_t}$, representing the content of a local store buffer for this variable in each thread t, (ii) a buffer counter variable x_{cnt_t} that records the current position in the store buffer of X in thread t.

The x86 TSO model maintains a single local buffer per process. This buffer is updated with stores to *any* of the global variables. However, we need additional vari-

ables to capture information about which global variable is stored in the buffer. The lhs variables contain the index of which global variable is addressed for each buffer element. The other variables are similar to PSO: (i) k *local* variables for each thread t: $lhs_{1_t}, \ldots, lhs_{k_t}$, representing the index of the global variable stored at a local store buffer in each thread t, (ii) k *local* variables for each thread t: $rhs_{1_t}, \ldots, rhs_{k_t}$, representing the value content of a local store buffer in each thread t, (iii) a buffer counter variable cnt_t that records the current position in the store buffer of thread t.

Step 3: Discover new predicates for P_M After verifying P under SC and constructing P_M, the remaining challenge is to discover a sufficient set of predicates for verifying that P_M satisfies a given specification. One of our main insights is that for buffered memory models such as x86 TSO and PSO, the predicates (and boolean program) used for verifying P under SC can be automatically leveraged to enable verification of P_M. This step corresponds to the "Extrapolate" box. This step takes as input the set of predicates V that were successful for verifying P under SC and outputs a new set V_M.

Predicates for the motivating example Next, we illustrate via our running example how to generate new predicates under PSO (the process under x86 TSO is similar).

Consider again the ABP algorithm of Fig. 1 and the 8 predicates listed earlier in Section 2.1. Following the structure of the additional variables in P_M, we can introduce additional predicates by cloning each predicate over a global variable X into new predicates over store-buffer variables $x_{1_t}, \ldots, x_{k_t}$. For example, assuming $k = 1$, in addition to $Msg = 0$, we introduce $Msg_1_t0 = 0$.

To keep track of the buffer size for each buffered variable, we introduce additional predicates. For instance, for a global variable Msg, to register possible values for Msg_cnt_t0, assuming $k = 1$, we introduce $Msg_cnt_t0 = 0$ and $Msg_cnt_t0 = 1$. Note that we do not introduce predicates such as $Msg_cnt_t1 = 0$ and $Msg_1_t1 = 0$ as thread 1 always accesses Msg via main memory. Another predicate we could have added is $Msg_1_t0 = Msg$. This predicate is not needed for the verification of ABP, however, we observe that such predicates can greatly reduce the state space of the model checker. In Section 5, we discuss rules and optimizations for generating new predicates for the PSO memory model and in Appendix B we present the details for the x86 TSO model (which are similar).

Finally, to ensure soundness, we add a designated flag $overflow$ to track when the buffer size grows beyond our predetermined bound k. Overall, with a bound of $k = 1$, from 8 predicates used for sequential consistency we generate 15 predicates for PSO:

$$(Msg = 0), (Ack = 0), (lSSt = 0), (lAck = 0), (lMsg = 0), (lRSt = 0), (lRCnt = lSCnt),$$
$$((lRCnt + 1) = lSCnt), (Msg_cnt_t0 = 0), (Msg_cnt_t0 = 1), (Ack_cnt_t1 = 0),$$
$$(Ack_cnt_t1 = 1), (Msg_1_t0 = 0), (Ack_1_t1 = 0), (overflow = 0)$$

Cube Search Space A critical part of the predicate abstraction algorithm is finding the weakest disjunction of cubes that implies a given formula (see [12, 6] for details). This is done by exploring the cube search space, typically ordered by cube size. Because the search is exponential in the number of predicates, most previous work on predicate abstraction has bounded the cube space by limiting the maximal cube size to 3.

The translation of SC predicates to PSO predicates implies a polynomial (in the buffer limit k, number of threads and number of shared variables) growth in the number

of predicates. Unfortunately, since cube search space is exponential in the number of predicates, exploration of the PSO cube space can sometimes be prohibitively expensive in practice.

For example, for ABP running on sequential consistency, the total number of cubes is $3^8 = 6561$. Here, 8 is the maximal sized cube which is the same as the number of predicates. And 3 means that we can use the predicate directly, its negation or the predicate can be absent. However, for PSO, the total number of cubes is 3^{15} exceeding 14 million cubes! If we limit cube size to 4, the SC cube search space becomes bounded by $\Sigma_{i=1}^{4} 2^i \binom{8}{i} = 288$, and the PSO cube space to be explored becomes bounded by $\Sigma_{i=1}^{4} 2^i \binom{15}{i} = 25630$.

The situation is worsened as the cube space is explored for every abstract transformer computation. Further, while in previous work, which mostly targets sequential programs, limiting the cube size to 3 seemed to work, with concurrency, where one needs to capture correlations between different threads, it is possible that we need a cube size of 4 or greater. As the number of predicates increases, directly exploring cubes of size 4 or more, even with standard optimizations, becomes infeasible (our experiments confirm that).

Reducing the PSO Cube Space using SC Cubes One of our main insights is that we can leverage the boolean program (the proof) under sequential consistency to simplify reasoning under relaxed memory models. Technically, we realize this insight by using the cubes from the boolean program under SC in order to guide the search in the cube space under the weak memory model.

In Fig. 2(b), this step is denoted under Cube Extrapolation where in addition to the steps in Predicate Extrapolation, we also extract the cubes that appear in the boolean program of P under SC.

For example, for ABP, we examine the boolean program $\mathcal{BP}(ABP, V)$ where V are the eight predicates listed earlier in Section 2.1, and observe the following cubes:

$$\begin{aligned} c_1 &= (lSSt = 0) \wedge (lAck = 0) \\ c_2 &= (lSSt = 0) \wedge \neg(lAck = 0) \\ c_3 &= (lMsg = 0) \wedge \neg(lRSt = 0) \\ c_4 &= \neg(lSSt = 0) \wedge \neg(lAck = 0) \\ c_5 &= \neg(lSSt = 0) \wedge (lAck = 0) \end{aligned}$$

Since these cubes do not use the two global variables Msg and Ack, it stands to reason that the same cubes would be obtained from the PSO cube space exploration, which is indeed the case.

In this simple example, the above cubes from the predicate abstraction under SC could be used directly for the predicate abstraction under PSO, *without* needing to search for these cubes. Of course, there are cases where SC cubes do contain buffered variables (such as Msg or Ack) and in that case we need to extrapolate from these cubes in order to obtain useful cubes under PSO (see Section 5).

Building the Boolean Program An enabling factor with cube extrapolation is that it changes the way we build the boolean program. We no longer require exponential search over the cube search space. In Section 4, we show how to use the cubes under SC as constraints over the cube space to reduce the size of the cube space to be explored under

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

if $x_{cnt_t} = k$ **then**
  abort("overflow")
 	
$x_{cnt_t} = x_{cnt_t} + 1$
 	
if $x_{cnt_t} = 1$ **then**
  $x_{1_t} = r$
...
if $x_{cnt_t} = k$ **then**
  $x_{k_t} = r$
	if $x_{cnt_t} = 0$ **then**
	  $r = X$
	if $x_{cnt_t} = 1$ **then**
	  $r = x_{1_t}$
	...
	if $x_{cnt_t} = k$ **then**
	  $r = x_{k_t}$
		▷ for each
		$X \in Gvar$
		generate:
		
		assume $(x_{cnt_t} = 0)$
		
		▷ end of generation
			while * **do**
			▷ for each $X \in Gvar$ generate:
			if $x_{cnt_t} > 0$ **then**
			if * **then**
			$X = x_{1_t}$
			if $x_{cnt_t} > 1$ **then**
			$x_{1_t} = x_{2_t}$
			...
			if $x_{cnt_t} = k$ **then**
			$x_{(k-1)_t} = x_{k_t}$
			
			$x_{cnt_t} = x_{cnt_t} - 1$
			▷ end of generation

Fig. 5. PSO Translation Rules: each sequence of statements is atomic

the weak memory model. Technically, the idea is to lift the cubes under SC to buffered counterparts by a translation similar to the way in which we extrapolated predicates under SC (described above).

We can then pack the extrapolated cubes as new predicates provided to the predicate abstraction procedure for the weak memory model, and limit cube size to 1. Limiting the maximal cube size to 1 turns the process of cube search from exponential to linear (in the number of predicates and cubes obtained from extrapolation). This is key to making predicate abstraction tractable for relaxed memory models.

In Fig. 2(b), to reduce clutter, both Predicate Extrapolation and Cube Extrapolation lead to the same Predicate Abstraction box. However, it is important to note that the process of predicate abstraction for Cube Extrapolation *does not* perform exponential search while the process for predicate extrapolation does. As we will see in the experimental results, Predicate Extrapolation is sufficient for simpler programs, while Cube Extrapolation is necessary for more complex programs.

Step 4: Model checking Once the boolean program is built (either via Predicate or Cube Extrapolation), the final step is to model check the program. This step is exactly the same as in the standard case of traditional predicate abstraction shown in Fig. 2(a).

3 Reduction

In this section, we describe a translation that transforms a program running on a relaxed memory model into a program with no weak memory model effects. The basic idea is to translate the store buffers in the semantics into variables that are part of the program. This translation enables us to leverage classic program analysis techniques such as predicate abstraction. Further, because the translation is parametric on the size of the buffer, it allows us to tailor the abstraction to the buffer size required by the particular concurrent algorithm. We show the process for the PSO memory model, the process for x86 TSO is similar and is shown in Appendix B

Reduction: PSO to SC The reduction takes as input a thread identifier (whose statements are to be handled), as well as a bound on the maximum buffer size k for the per-variable buffer. Here, k denotes the maximum number of writes to global variables without a fence in-between the writes. While the translation presented here uses a fixed k for all global variables, we can easily use different k's for different variables.

The translation considers in turn every statement that involves global variables. We introduce a translation function, which takes as input a statement, a thread identifier, and a bound on the maximum buffer size and produces a new statement as output:

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

As a shorthand we write $[\![\text{S}]\!]^t_k$ for $[\![\text{S}, t, k]\!]$. $[\![\text{S}]\!]^t_k$ denotes the statement obtained from translating statement S in thread t with buffer bound k.

To perform the translation, the entries of the per-variable buffer are translated into thread-local variables and a local counter is introduced to maintain its depth. That is, for each global variable X and thread t, we introduce the following local variables:

- buffer content variables: x_{1_t},...,x_{k_t}, where k is the maximum size of the buffer.
- a buffer counter variable: x_{cnt_t}.

Fig. 5 presents the translation of the three program code statements and the memory subsystem statement (flush). In the translation, the newly generated sequence of statements is atomic.

Store to a global variable $[\![\boldsymbol{X = r}]\!]^t_k$***:*** The store to a global variable X first checks if we are about to exceed the buffer bound k and if so, the program aborts. Otherwise, the counter is increased. The rest of the logic checks the value of the counter and updates the corresponding local variable. The global variable X is not updated and only local variables are involved.

Load from a global variable $[\![\boldsymbol{r = X}]\!]^t_k$***:*** 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., $x_{cnt_t} = 0$), the load is performed directly from the global store. We do not need to check whether the buffer limit k is exceeded as that is ensured by the global store.

Fence statement $[\![\textbf{\textit{fence}}]\!]^t_k$***:*** For *each* shared variable X, the fence statement waits for the buffer of X to be empty (flush instructions to be executed). The fence has no effect on X.

Flush action $[\![\textbf{\textit{flush}}]\!]^t_k$***:*** The flush action is translated into a loop with a non-deterministic exit condition (we use $*$). New statements are introduced for each global variable X. If the buffer counter for the variable is positive, then it non-deterministically decides whether to update the global variable X or to continue the iteration. If it has decided to update X, the earliest write (i.e. x_{1_t}) is stored in X. The contents of the local variables are then updated by shifting: the content of each x_{i_t} is taken from the content of the successor $x_{(i+1)_t}$ where $1 \leq i < k$. Finally, the buffer count is decremented. The composite statement inside the while loop is generated for each global variable. To ensure a faithful translation of the flush action, the whole newly generated statement is placed after *each* statement of the resulting program. The atomic statements are translated directly, without change (not shown in the figure).

The translation extends naturally to a sequence of statements and to programs with n concurrent threads: $[\![P]\!]_k = [\![S]\!]_k^1 \parallel \cdots \parallel [\![S]\!]_k^n$, leading to the following theorem:

Theorem 1 (Soundness of Translation). *For a given program, P and a safety specification S, if $P \not\models_{pso} S$ then there exists a $k \in \mathbb{N}$ such that $[\![P]\!]_k \not\models_{sc} S$.*

From the theorem it follows that if $[\![P]\!]_k \models_{sc} S$ then $P \models_{pso} S$. When we successfully verify the program with a given k, it is guaranteed that no execution of the program ever requires a buffer of size larger than k. If the program does have an execution which exceeds k, then during verification we will encounter overflow and can attempt a higher value of k. That is, if we verify the program for a certain bound k, then the algorithm is correct for any size of the buffer greater or equal to k. In our experience, most concurrent algorithms exhibit low values for k as typically they use fences after a small number of global stores.

4 Predicate Abstraction for Relaxed Memory Models

In this section we describe how predicate abstraction is used to verify concurrent programs running on relaxed memory models. The central point we address is how to discover the predicates necessary for verification under the relaxed model from the predicates and the proof that was successful for verification of the program under sequential consistency (SC).

4.1 Predicate Abstraction

Predicate abstraction [15] is a special form of abstract interpretation that employs cartesian abstraction over a given set of predicates. Given a program P, and vocabulary (set of predicates) $V = \{p_1, \ldots, p_n\}$ with corresponding boolean variables $\hat{V} = \{b_1, \ldots, b_n\}$, predicate abstraction constructs a boolean program $\mathcal{BP}(P, V)$ that conservatively represents behaviors of P using only boolean variables from $\hat{V}$ (corresponding to predicates in V). We use $[b_i]$ to denote the predicate p_i corresponding to the boolean variable b_i. We similarly extend $[b]$ to any boolean function b.

Next we explain how to construct $\mathcal{BP}(P, V)$. A literal is a boolean variable or its negation. A *cube* is a conjunction of literals, the size of a cube is the number of literals it contains. The concrete (symbolic) domain is defined as formulae over the predicates $p_1, \ldots, p_n$. The abstract domain is a disjunctions of cubes over the variables $b_1, \ldots, b_n$. The abstraction function α maps a formula φ over predicates from V to the weakest disjunction of cubes d such that $[d] \Rightarrow \varphi$.

The abstract transformer of a statement st w.r.t. a given vocabulary V can be computed using weakest-precondition computation and performing implication checks using a theorem prover:

$$b_i = choose(\alpha(wp(st, p_i)), \alpha(wp(st, \neg p_i,)))$$

where

$$choose(\varphi_t, \varphi_f) = \begin{cases} 1, \ \varphi_t \text{ evaluates to true;} \\ 0, \text{ only } \varphi_f \text{ evaluates to true;} \\ \star, \text{ otherwise.} \end{cases}$$

Different predicate abstraction techniques use different heuristics for reducing the number of calls to the prover.

```
Input: Vocabulary V, Statement st, Maximum cube size k
Output: Abstract transformer for st over predicates from V

function computeTransformer(V, st, k)
  for each p ∈ V do
    ψp+ = ψp- = false
    φ+ = wp(st, p)
    φ- = wp(st, ¬p)
    for each i = 1 ... k do
      cubes+ = buildBoundedCubes(V, i, ψp+)
      if cubes+ = ∅ then break
      ψp+ = computeApprox(cubes+, φ+, ψp+)
    for each i = 1 ... k do
      cubes- = buildBoundedCubes(V, i, ψp-)
      if cubes- = ∅ then break
      ψp- = computeApprox(cubes-, φ-, ψp-)
    ψ(p) = choose(ψp+, ψp-)
```

(a)

```
Input: RMM predicates V_rmm, RMM cubes C_rmm, Statement st,
Output: Abstract transformer for st over predicates from V_rmm ∪ C_rmm

function computeTransformer(V_rmm, C_rmm, st)
  for each p ∈ V_rmm do
    ψp+ = ψp- = false

    φ+ = wp(st, p)
    ψp+ = computeApprox(V_rmm ∪ C_rmm, φ+, ψp+)

    φ- = wp(st, ¬p)
    ψp- = computeApprox(V_rmm ∪ C_rmm, φ-, ψp-)

    ψ(p) = choose(ψp+, ψp-)
```

(b)

Fig. 6. Computing abstract transformers: (a) classical predicate abstraction; (b) using extrapolation. COMPUTEAPPROX is shown in Fig. 7.

```
function computeApprox(cubes, φ, ψ)
  for each c ∈ cubes do
    if c ⇒ φ then
      ψ = ψ ∨ c
  return ψ
```

Fig. 7. Predicate abstraction - helper function.

Fig. 6 (a) shows a standard predicate abstraction algorithm in the spirit of [6]. The algorithm takes as input a statement st and a set of predicates (vocabulary) V. It then computes an abstract transformer for st using combinations of predicates from V. The algorithm works by computing an update formula $\psi(p)$ for every predicate p. The update formula is constructed as a choice between two sub-formulae, ψ_p^+ which holds when p should be set to $true$, and ψ_p^- which holds when p should be set to $false$.

The function BUILDBOUNDEDCUBES builds cubes of size i over predicates from V, checking that cubes of size i are not subsumed by previously generated cubes in ψ_p^+ or ψ_p^-. This function is standard, and we do not list it here due to space restrictions.

There are other algorithms that can be used here, such as the Flanagan&Qadeer's [14], or the one of Das et al. [10]. However, our focus is not on these optimizations, but on leveraging information from the verification of the SC program to discover the new predicates for verification under the relaxed memory model.

4.2 Predicate Extrapolation: from predicates under SC to predicates under relaxed model

Given a program which successfully verified under SC, our first approach to verifying the program under x86 TSO or PSO is to:

- Reduce the program as described in Section 3. That is, given a statement st of the original program, obtain a new statement st_{pso} or st_{tso} from the translation.
- Compute $V_{pso} = \text{EXTRAPOLATEPSO}(V)$ for PSO as discussed in Section 5, or for TSO: $V_{tso} = \text{EXTRAPOLATETSO}(V)$ (the TSO extrapolation is discussed in Appendix B and is similar to PSO). This extrapolates from the set of input predicates under SC and derives new predicates under x86 TSO or PSO.
- Invoke $\text{COMPUTETRANSFORMER}(V_{pso}, st_{pso}, k)$ to build the abstract transformer under PSO. Similarly, invoke $\text{COMPUTETRANSFORMER}(V_{tso}, st_{tso}, k)$ for x86 TSO. Here, k is the appropriate buffer bound. The function COMPUTETRANSFORMER is shown in Fig. 6 (a).

That is, with this approach, the entire predicate abstraction tool chain remains the same except we change the input to the function for computing abstract transformers to be the new predicates V_{pso} and the new statement st_{pso}.

4.3 Cube Extrapolation: from SC proof to PSO predicates

As we will see in our experimental results, predicate extrapolation is effective only in some cases. The problem is that on more complex programs, the cube search space increases significantly meaning the function COMPUTETRANSFORMER as described in Fig. 6 (a) times out. Next, we discuss another approach for computing abstract transformers under the relaxed memory model.

Core Idea The core idea is that cubes generated during SC predicate abstraction capture invariants that are important for correctness under SC, but the same relationships between variables can be extrapolated to relationships between variables in the relaxed setting. Based on this observation, we extrapolate from these cubes similarly to the way we extrapolated from the predicates under SC. We use the function $C_{pso} = \text{EXTRAPOLATEPSO}(C)$ where C denotes the cubes under SC. The newly generated extrapolated cubes C_{pso} are then used as predicates for the verification.

The key point is that the cube search over C_{pso} is now limited to cubes of size 1! The steps are as follows:

- Compute $V_{pso} = \text{EXTRAPOLATEPSO}(V)$
- Compute $C_{pso} = \text{EXTRAPOLATEPSO}(C)$
- Invoke $\text{COMPUTETRANSFORMER}(V_{pso}, C_{pso}, st)$ as shown in Fig. 6 (b) and taking as input extrapolated predicates and extrapolated cubes together with the statement.

The process for the x86 TSO model is identical.

Search vs. Extrapolation In contrast to the standard COMPUTETRANSFORMER of Fig. 6 (a), the algorithm of Fig. 6 (b) *does not* perform exhaustive search of the cube space. In particular, it does not take as input the parameter k. That is, our new way of building transformers is based on extrapolating from a previous (SC) proof.

5 Extrapolating Predicates: SC to PSO

In this section, we elaborate on how the function $preds_{pso} = \text{EXTRAPOLATE}\text{PSO}(preds_{sc})$ operates. The operation $\text{EXTRAPOLATE}\text{TSO}(preds_{sc})$ for TSO is similar and is discussed in Appendix B. The function EXTRAPOLATEPSO computes the ingredients, that is, the new predicates $preds_{pso}$, using which the final abstraction is built. We discuss the core reasons for introducing the new predicates. This reasoning is independent of predicate abstraction and can be used with other verification approaches.

Any abstraction for store buffer based RMMs such as PSO must be precise enough to preserve the following properties: (i) Intra-thread coherence: If a thread stores several values to shared variable X, and then performs a load from X, it should not see any value it has itself stored except the most recent one. (ii) Inter-thread coherence: A thread T_i should not observe values written to shared variable X by thread T_j in an order different from the order in which they were written. (iii) Fence semantics: If a thread T_i executes a fence when its buffer for variable X is non-empty, the value of X visible to other threads immediately after the fence should be the most recent value T_i wrote.

Fig. 13 shows a simple example in which a single thread stores two values into a shared variable X and then loads the value of X into $l1$. To successfully verify this program, the abstraction we use must be precise enough to capture intra-thread coherence.

Thread 1:

```
1  X=0;
2  X=1;
3  l1=X;
4  fence;
5  assert (X = l1);
```

Fig. 8. Intra-thread coherence example

5.1 Generic Predicates

For a shared variable X, if we use a buffer with a max size of 1, our translation adds the predicates: $X_cnt_t1 = 0, X_cnt_t1 = 1$ indicating the last location of the buffer for X which has been written to by thread 1, but not yet flushed. These predicates serve multiple purposes: (i) track the store buffers size; (ii) provide knowledge during `store` and `load` operations on where to write/read the value of X. (iii) preserve *Intra-thread coherence* in the abstraction.

Another predicate we generate is: $overflow = 0$. This predicate supplements the previously described predicates, giving indication of when the number of subsequent stores to a shared variable X, without a fence or a flush in between these stores, exceeds the limit k of our abstraction. This is crucial to ensure soundness of the abstraction.

The general extrapolation rule, which is independent of the verified program and of the input SC predicates $preds_{sc}$, is:

Rule 1 *For a buffer size bound k, add the following predicates to* $preds_{pso}$*:*

- $\{\mathbf{V}_cnt_\mathbf{T} = i \mid 1 \leq i \leq k, \mathbf{V} \in Gvar, \mathbf{T} \in Thread\}$
- $overflow = 0$

5.2 Extrapolating from $preds_{sc}$

We now describe how to extrapolate from the predicates in the set $preds_{sc}$ in order to compute new predicates that become part of $preds_{pso}$. The rule below ensures that the SC executions of the new program can be verified.

Rule 2 *Update the set* $preds_{pso}$ *to contain the set* $preds_{sc}$

Next, we would like properties on the values of a shared variable X captured by predicates in $preds_{sc}$ to also be captured for the buffered values of X. For example if $preds_{sc}$ contains $X = 0$, we add the predicate $X_1_t1 = 0$ for a buffer of X for thread T_1. This can be seen in the example of Fig. 13 where we need to track that the buffered value of X is 0 at line 1. We summarize these observations in the following rule:

Rule 3 *Update* $preds_{pso}$ *to contain the set* $\bigcup_{p_{sc} \in preds_{sc}} lift(p_{sc})$

Here, $lift(p_{sc})$ generates from each SC predicate a set of PSO predicates where the original variables are replaced with buffered versions of the variables (for each buffered version of a variable and their combination).

In addition to the above rules, adding a predicate $X_1_t1 = X$ ensures that the shared value of X and the buffered value of X are in agreement (when the predicate is set to $true$). This reduces the time and space of model checking. Following similar reasoning, the predicate $X_1_t1 = X_2_t1$ is also added.

Rule 4 *For* $\mathbf{V} \in Gvar$*,* $\mathbf{T} \in Thread$ *and* k *the buffer bound, update* $preds_{pso}$ *to contain the sets:*

- $\{\mathbf{V}_(i-1)_\mathbf{T} = \mathbf{V}_i_\mathbf{T} \mid 2 \leq i \leq k\}$
- $\{\mathbf{V}_i_\mathbf{T} = \mathbf{V} \mid 1 \leq i \leq k\}$

The above rules add both: generic predicates that are independent of $preds_{sc}$ as well as predicates that are extrapolated from $preds_{sc}$. But these rules may sometimes generate a larger set of predicates than necessary for successful verification. We now describe several optimizations that substantially reduce that number.

Rule 5 ***Read-only shared variables*** *If a thread* t *never writes to a shared variable* X *do not extrapolate the SC predicates referencing* X *to their PSO counterparts for* t*.*

Rule 6 ***Predicates referencing a shared variable more than once*** *Replace all occurrences of the shared variable with the same buffered location.*

For example, for $X{\leq}Y \wedge 0{\leq}X$, where X is referred to more than once, we generate the new predicate $X_1_t1{\leq}Y \wedge 0{\leq}X_1_t1$, but we do not generate the predicate $X_1_t1{\leq}Y \wedge 0{\leq}X_1_t2$. The intuition is that the SC predicate captures information regarding the value of X at some point in the execution and when extrapolating the predicate to PSO, we need to capture that information regarding the shared value of X or its buffered value, yet the mixture of the two is redundant. Similarly for $Y2 \leq Y1$, we do not necessarily need to generate the predicate $Y2_1_t2 \leq Y1_1_t1$.

Rule 7 ***Predicates referencing different shared variables*** *For a predicate referencing more than one shared variable, if it can be guaranteed that a fence will be executed between every two shared location writes , restrict to generating predicates that relate to one buffered location at most.*

In summary, EXTRAPOLATEPSO is computed by applying the above seven rules for the predicates in V and C (both obtained from the verification under SC).

6 Experimental Evaluation

We implemented predicate abstraction based on predicate extrapolation (PE) and cube extrapolation (CE) as outlined earlier in our tool called CUPEX. Then, we thoroughly evaluated the tool on a number of challenging concurrent algorithms. All experiments were conducted on an Intel(R) Xeon(R) 2.13GHz with 250GB RAM. The key question we seek to answer is whether predicate extrapolation and cube extrapolation are precise and scalable enough to verify all of our (relaxed) programs.

6.1 Prototype Implementation

CUPEX works in two phases. In the first phase, given an input program, it applies abstraction and produces a boolean program. In the second phase, the boolean program is verified using a three-valued model checker for boolean programs. To reduce the cube search space, CUPEX uses optimizations such as bounded cube size search and cone of influence. For every assignment statement in the original program, it updates only the boolean variables corresponding to predicates which contain the assigned variable from the statement (subject to aliasing). The search in the cube space is performed in increasing cube size order, thus we find the weaker (smaller) cubes first. CUPEX uses Yices 1.0.34 as the underlying SMT solver.

The second phase relies on a three-valued model checker to verify the boolean program. Our model checker uses 3-valued logic for compact representation, and handles assume statements in a number of clever ways, performs partial concretization of assume conditions and merges states after updates.

6.2 Results

We evaluated CUPEX on the following concurrent algorithms: Dekker's mutual exclusion algorithm [11], Peterson's mutual exclusion algorithm [25], Szymanski mutual exclusion algorithm [28], Alternating Bit protocol (already discussed in Section 2), an Array-based Lock-Free Queue (here, we verified its memory safety), Lamport's Bakery algorithm [23] and the Ticket locking algorithm [3]. The first three algorithms are finite-state, while the last four are infinite-state. For each algorithm, we evaluated our tool for x86 TSO and PSO models. We ran tests with buffer bounds ranging from $k \in 1 \dots 3$. For each k, we tested various fence configurations. We present in the result tables values for $k = 1$, obtained for the minimal fence configurations which successfully verified.

Meaning of table columns Our results are summarized in Table 1 and Table 2. The meaning of most table columns is self explanatory, but we elaborate on the following columns of Table 1:

- Build Boolean Program (i) # input preds: number of initial input predicates. For x86 TSO and PSO, these are obtained by extrapolating from the predicates in the # input preds column of the corresponding SC program. (ii) # SMT calls: total number of calls (in thousands) to the SMT solver required to build the boolean program (BP). (iii) time: time in seconds that it took to build the boolean program. We use T/O for timed out (keeps running after 10 hours). (iv) # cubes used: total

Table 1. Results for Predicate Extrapolation.

		Build Boolean Program					Model check		
algorithm	memory model	# input preds	# SMT calls (K)	time (sec)	# cubes used	cube size	# states (K)	memory (MB)	time (sec)
Dekker	SC	7	0.7	0.1	0	1	14	6	1
	PSO	20	26	6	0		80	31	5
	TSO	18	22	5	0		45	20	3
Peterson	SC	7	0.6	0.1	2	2	7	3	1
	PSO	20	15	3	2		31	13	3
	TSO	18	13	3	2		25	11	2
ABP	SC	8	2	0.5	5	2	0.6	1	0.6
	PSO	15	20	4	5		2	3	1
	TSO	17	23	5	5		2	3	1
Szymanski	SC	20	16	3.3	1	2	12	6	2
	PSO	35	152	33	1		61	30	4
	TSO	37	165	35	1		61	31	5

number of cubes in the boolean program whose size is greater than 1, that is, these are cubes composed of 2 or more input predicates. (v) cube size: maximum cube size found in the boolean program. For instance, cube size 4 means that there exist cubes which combine 4 input predicates.

- Model check (i) # states: total number of states (thousands) explored by the underlying three-valued model checker.

Table 2 contains two additional columns in the Build Boolean Program column: (i) method used to build the boolean program: PE , CE or, for SC programs where no new predicates are generated, Trad (Traditional). (ii) # input cubes: These are obtained by extrapolating from the cubes in the # cubes used column of the corresponding SC program. These cubes are then added to the initial # input preds and CE is performed.

6.3 Observations

Sequential Consistency We assume that the predicates for the original SC program are provided by the programmer or inferred using existing SC verification tools such as [16]. For our benchmarks we manually provided the SC predicates, since we focus on the relaxed memory verification.

Predicate Extrapolation We first focus on the results of Table 1. Here, PE was quick enough to verify all programs under x86 TSO and PSO. For example for Dekker's algorithm, even though there was a significant increase in the number of predicates (7 input predicates required to verify the program under SC yet under PSO, 20 predicates were needed), the newly generated predicates were precise enough to prove the program. A similar pattern can be observed for Peterson, ABP and Szymanski. For all four algorithms, a small cube size (at most 2) was sufficient for verification. Furthermore, the number of cubes of size 2 that are used in the boolean program is fairly small. Overall, these four programs require a small cube size to verify, at most 2, leading to quick building of the boolean program. For these programs, CE was unnecessary.

Table 2. Results for Predicate Extrapolation and Cube Extrapolation

		Build Boolean Program							Model check		
algorithm	memory model	method	# input preds	# input cubes	# SMT calls (K)	time (sec)	# cubes used	cube size	# states (K)	memory (MB)	time (sec)
Queue	SC	Trad	7	-	20	5	50	4	1	2	1
	PSO	PE	15	-	5,747	1,475	412		1	4	1
		CE		99	98	17	99		11	6	2
	TSO	PE	16	-	11,133	2,778	412		12	4	1
		CE		99	163	31	99		12	7	2
Bakery	SC	Trad	15	-	1,552	355	161	4	20	8	2
	PSO	PE	38	-	-	T/O	-		-	-	-
		CE		422	9,018	1,773	381		979	375	104
	TSO	PE	36	-	-	T/O	-		-	-	-
		CE		422	7,048	1,386	383		730	285	121
Ticket	SC	Trad	11	-	218	51	134	4	2	2	1
	PSO	PE	56	-	-	T/O	-		-	-	-
		CE		622	15,644	2,163	380		193	123	40
	TSO	PE	48	-	-	T/O	-		-	-	-
		CE		622	6,941	1,518	582		71	67	545

Cube Extrapolation Next, we discuss the results of Table 2. We first observe that for Queue, under PSO, PE required around 6 million calls to the SMT solver and took a little over 24 minutes to complete in building the boolean program. Indeed, the combination of increased cube size (4) together with 15 initial input predicates significantly affected the running time for building the boolean program. Interestingly, we observe that CE, was able to reduce the number of calls to the SMT solver by a factor of 60 and reduce the running time by a factor of 80. Importantly, CE was precise enough to verify the program. Here we see that CE generated 99 new cubes which were extrapolated from the 50 SC cubes. The final program used exactly these 99 cubes, meaning that CE did not generate redundant cubes.

For both Bakery and Ticket, the benefits of CE are even more startling. With PE, building the boolean program fails under both x86 TSO and PSO due to a time out. However, CE massively reduced the number of SMT calls enabling successful generation of the boolean program. The set of cubes CE returned was fairly close to the set of cubes used during the boolean program verification. For instance, in Bakery under PSO, 422 input cubes were generated out of which 381 were used in the boolean program (fairly close to ideal).

It is worth noting that in all benchmarks we experimented on, the minimal fence placement required was different for x86 TSO and PSO.

Discussion Next we note two observations from our approach which we encountered experimentally and which we believe are interesting items for future work.

First, when we directly applied CE to the Ticket algorithm, it took hours to verify for PSO. To solve this problem, we hypothesized that given a safety property, which does not reference buffered values, we may allow inconsistent values at buffered locations, and that inconsistency would be resolved when those values are flushed and before an

error state is reached. Therefore, to enable quicker verification, we first applied CE as usual, and then automatically removed all predicates referencing buffered values from the resulting cubes found in the boolean program after CE. Such a reduction preserves soundness while abstracting the proof. We note that although this approach worked for Ticket under PSO, when we tried it under x86 TSO this additional pass introduced too much imprecision and the program failed to verify (the table reports results for Ticket on PSO using this additional pass and on x86 TSO without this pass).

Second, for the Queue algorithm our initial successful SC proof was insufficient to extrapolate from. Portions of the program where the boolean program under SC lost precision due to abstraction were amplified by the extrapolation. For instance, where the SC proof used a predicate $Tail < Head$ which was unknown through parts of the SC proof with no adverse effects, the extrapolated proof propagated this uncertainty causing an error state to be reached. Adding $Tail \leq Head$ strengthened the SC proof and enabled successful extrapolation (this is the result we report in the Table).

Summary For cubes of small size, 2 or less, with PE, CUPEX builds the boolean program quickly and is precise enough to verify the program. For larger cube sizes, PE takes too long to build the boolean program or times out. However, CUPEX with CE enables us to build the boolean program in reasonable time and critically, is precise enough to verify the program both for x86 TSO and PSO.

7 Related Work

There has been almost no work on automatically verifying *infinite-state* concurrent programs running on relaxed memory models. We briefly survey some of the more closely related work.

Model Checking for Relaxed Memory Models The works of [24, 20, 21, 18] describe explicit-state model checking under several memory models. In [7], instead of working with operational memory models and explicit model-checking, they convert programs into a form that can be checked against an axiomatic model specification. These approaches do not handle infinite-state programs. The work in [22] focuses on programs that are finite-state under SC but infinite-state under x86 TSO and PSO and suggests an abstraction to deal with the issue. Unfortunately, it also cannot handle general infinite-state programs (i.e., the program must be finite-state under SC).

The works of [2, 5] present a code-to-code transformation which encodes the relaxed memory semantics into the program. Our approach goes beyond this transformation and tackles the difficulty of verifying the newly obtained relaxed program. These new programs are more difficult to verify because of the complexity added by the encoded semantics. Our approach solves this problem by learning from the proof under sequential consistency.

The work of [1] builds on [22] and handles infinite state programs (on x86 TSO) by applying both predicate abstraction and store buffers abstraction. Their approach discovers predicates via traditional abstraction refinement and does not reuse information from the proof under SC, while in our approach we present a technique which leverages an existing proof under SC in order to derive a new proof for a more relaxed program.

Further, we also handle a memory model (PSO) that allows for more behaviors and complexity than x86 TSO.

Lazy abstraction The work of [17] introduces the concept of adjusting the level of abstraction for different sections of the verified program's state space. This is achieved by applying on-the-fly refinement for search-tree sub-graphs. Their approach does not construct a boolean program during verification. However, encoding the weak memory semantics in the code and extrapolating from the SC proof are concepts applicable for extending lazy abstraction to relaxed memory models. The backwards counter-example analysis phase, which requires costly calls to the theorem prover, may in part be avoided by anticipating in each branch of the search tree which predicates are required.

8 Conclusion and Future Work

We introduced a novel approach for predicate abstraction of concurrent programs running on relaxed memory models such as x86 TSO and PSO. The essence of our approach is extrapolation: learning from an existing proof of the program under sequential consistency in order to obtain a proof for a more relaxed version of the program.

We implemented our extrapolation approach and successfully applied it to automatically verify several challenging concurrent algorithms for both x86 TSO and PSO. This is the first time some of these programs have been verified for a model as relaxed as PSO.

As future work, we plan to investigate how these techniques apply to other relaxed models, both hardware models such as Power, as well as software programming models such as [8].

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. In *SAS* (2012), A. Miné and D. Schmidt, Eds., vol. 7460 of *Lecture Notes in Computer Science*, Springer, pp. 164–180.
2. ALGLAVE, J., KROENING, D., NIMAL, V., AND TAUTSCHNIG, M. Software verification for weak memory via program transformation. In *ESOP* (2013), M. Felleisen and P. Gardner, Eds., vol. 7792 of *Lecture Notes in Computer Science*, Springer, pp. 512–532.
3. ANDREWS, G. R. *Concurrent programming - principles and practice*. Benjamin/Cummings, 1991.
4. ATIG, M. F., BOUAJJANI, A., BURCKHARDT, S., AND MUSUVATHI, M. On the verification problem for weak memory models. In *POPL* (2010), M. V. Hermenegildo and J. Palsberg, Eds., ACM, pp. 7–18.
5. ATIG, M. F., BOUAJJANI, A., AND PARLATO, G. Getting rid of store-buffers in tso analysis. In *CAV* (2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *Lecture Notes in Computer Science*, Springer, pp. 99–115.
6. BALL, T., MAJUMDAR, R., MILLSTEIN, T. D., AND RAJAMANI, S. K. Automatic predicate abstraction of c programs. In *PLDI* (2001), M. Burke and M. L. Soffa, Eds., ACM, pp. 203–213.

7. Burckhardt, S., Alur, R., and Martin, M. M. K. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI* (2007), J. Ferrante and K. S. McKinley, Eds., ACM, pp. 12–21.
8. Burckhardt, S., Baldassin, A., and Leijen, D. Concurrent programming with revisions and isolation types. In *OOPSLA* (2010), W. R. Cook, S. Clarke, and M. C. Rinard, Eds., ACM, pp. 691–707.
9. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL* (1977).
10. Das, S., Dill, D. L., and Park, S. Experience with Predicate Abstraction. In *CAV* (1999).
11. Dijkstra, E. Cooperating sequential processes, TR EWD-123. Tech. rep., Technological University, Eindhoven, 1965.
12. Donaldson, A., Kaiser, A., Kroening, D., and Wahl, T. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *CAV* (2011).
13. Dubois, M., Scheurich, C., and Briggs, F. A. Memory access buffering in multiprocessors. In *ISCA* (1986).
14. Flanagan, C., and Qadeer, S. Predicate abstraction for software verification. In *POPL* (2002).
15. Graf, S., and Saïdi, H. Construction of abstract state graphs with PVS. In *CAV* (1997).
16. Gupta, A., Popeea, C., and Rybalchenko, A. Threader: A constraint-based verifier for multi-threaded programs. In *CAV* (2011).
17. Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Lazy abstraction. In *POPL* (2002).
18. Huynh, T. Q., and Roychoudhury, A. Memory model sensitive bytecode verification. *Form. Methods Syst. Des.* (2007).
19. IBM. *Power ISA v.2.05*. 2007.
20. Jonsson, B. State-space exploration for concurrent algorithms under weak memory orderings. *SIGARCH Comput. Archit. News* (2008).
21. Kuperstein, M., Vechev, M., and Yahav, E. Automatic inference of memory fences. In *FMCAD* (2010).
22. Kuperstein, M., Vechev, M., and Yahav, E. Partial-coherence abstractions for relaxed memory models. In *PLDI* (2011).
23. Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* (1974).
24. Park, S., and Dill, D. L. An executable specification and verifier for relaxed memory order. *IEEE Trans. on Computers 48* (1999).
25. Peterson, G. L. Myths about the mutual exclusion problem. *Inf. Process. Lett. 12*, 3 (1981).
26. Sarkar, S., Sewell, P., Nardelli, F. Z., Owens, S., Ridge, T., Braibant, T., Myreen, M. O., and Alglave, J. The semantics of x86-cc multiprocessor machine code. In *POPL* (2009).
27. SPARC International, Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
28. Szymanski, B. K. A simple solution to Lamport's concurrent programming problem with linear wait. In *ICS* (1988).

A Definitions

A.1 Language

$$
\begin{array}{lll}
l, l' \in Label & & \\
r \in Lvar & & \\
X, Y \in Gvar & & \\
\mathcal{B} \in Bexp & ::= & ... \\
\mathcal{N} \in Nexp & ::= & ... \\
NS \in NStmt & ::= & {}^{l}r = \mathcal{N} \mid {}^{l}r = X \mid {}^{l}X = r \mid {}^{l}\textbf{fence} \mid \\
 & & {}^{l}\textbf{goto}\ l' \mid {}^{l}\textbf{if}\ \mathcal{B}\ \textbf{then}\ NS\ \textbf{else}\ NS \mid NS; NS \\
S \in Stmt & ::= & NS \mid {}^{l}\textbf{begin_atomic}\ NS;\ {}^{l}\textbf{end_atomic} \mid \\
 & & {}^{l}\textbf{if}\ \mathcal{B}\ \textbf{then}\ S\ \textbf{else}\ S \mid S; S \\
P & ::= & S \parallel ... \parallel S
\end{array}
$$

Fig. 9. Language

We consider a basic parallel language shown in Fig. 9. Some parts of the language are standard and we do not elaborate on them further (e.g. how to form numerical expressions denoted by $Nexp$ or boolean expressions denoted by $Bexp$). To simplify translation (and without losing expressivity), we separate per-thread local variables (denoted by $Lvar$) from global shared variables (denoted by $Gvar$). Expressions, both numerical and boolean, can only refer to local variables and statements can read and write global variables. In addition, the language contains the statements for starting and ending atomic sections. We use $Label$ to denote the set of all labels and $Stmt$ to denote all statements (these consist of statements which do not contain atomics such as NS and others which can contain atomic primitives). Each statement in the program is uniquely labeled.

A transition system is a tuple $\langle \Sigma_0, \Sigma, T \rangle$, where Σ is the set of program states, $\Sigma_0 \subseteq \Sigma$ are the initial program states, and T is the transition relation: $T \subseteq \Sigma \times Thread \times Stmt \times \Sigma$.

A.2 Language and Relaxed Semantics

We use a standard parallel programming language with the usual definitions of statement and expression evaluation. For simplicity we assume that parallelism is created with parallel composition. To simplify the translation that follows in later sections (without losing expressivity), we separate per-thread local variables ($Lvar$) from shared variables ($Gvar$). Expressions can only refer to local variables and statements can read and write global variables. In addition, the language contains statements enabling atomic operations. We use $Label$ to denote the set of all labels and $Stmt$ to denote all statements. Each statement in the program is uniquely labeled. Under sequential consistency, the (interleaving) semantics of the language is standard. We next proceed with defining its semantics under PSO.

We use $tid \in Thread$ to denote a thread identifier from a finite set of thread identifiers, and D for the domain from which variables take values. A program state s is a tuple $\langle pc, L, G \rangle$, where:

- $pc \in Thread \to Label$ is a map of threads to program labels.
- $L \in Thread \to Lvar \to D$ is a map from threads to local variables to values.
- $G \in Gvar \to D$ is a map from shared variables to values.
- $in_atomic \in \{Thread \cup \bot\}$ maintains which thread is inside an atomic section ($\bot$ means the atomic section is free).

Semantics 1 Operational semantics under PSO.

$$\frac{pc = l \qquad {}^{l}r = X \qquad B(X) = \epsilon \qquad G(X) = v \qquad enabled(tid)}{L'(r) = v \qquad pc' = n(pc)} \ (\text{LOAD-G})$$

$$\frac{pc = l \qquad {}^{l}r = X \qquad B(x) = b \cdot v \qquad enabled(tid)}{L'(r) = v \qquad pc' = n(pc)} \ (\text{LOAD-B})$$

$$\frac{pc = l \qquad {}^{l}X = r \qquad B(x) = b \qquad L(r) = v \qquad enabled(tid)}{B'(x) = b \cdot v \qquad pc' = n(pc)} \ (\text{STORE})$$

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

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

$$\frac{pc = l \qquad {}^{l}begin_atomic \qquad in_atomic = \bot}{in_atomic = tid \qquad pc' = n(pc)} \ (\text{BEGIN-ATOMIC})$$

$$\frac{pc = l \qquad {}^{l}end_atomic \qquad in_atomic = tid}{in_atomic = \bot \qquad pc' = n(pc)} \ (\text{END-ATOMIC})$$

When updating mappings, we use $M'(x) = v$ as a shorthand for $M' = M[x \mapsto v]$. If the thread tid is clear from the context, we use pc for $pc(tid)$ and $n(pc)$ to denote the label following pc. To track whether a thread is enabled to take a step, we use the predicate $enabled(tid) = (in_atomic = \bot) \vee (in_atomic = tid)$. That is, a thread can only take a step if no other is inside the atomic section or the atomic section is free.

To model relaxed memory model effects, we augment the program state with a store buffer $B \in Thread \to Gvar \to D^*$ for PSO. That is, we keep a sequence (store buffer) of values for each (thread, global variable) pair. The semantics of the relevant language statements under the PSO memory model are given in Semantics 1. The semantics above is formalized in Appendix A.1

A.3 Standard (Naive) Predicate Abstraction under RMM is Unsound

We now consider the following scheme for predicate abstraction: (i) construct a boolean program directly from the program of Fig. 10 using standard predicate abstraction; (ii) execute the boolean program using PSO semantics. This scheme is simple, but unfortunately, it is unsound. We now show that following this scheme, we can produce a boolean program that successfully verifies our assertion. This is unsound because we know the existence of an assertion violation, namely the one shown in Fig. 11(a).

initial: X=Y=0

Thread 0:		Thread 1:
1	`X = Y+1`	1 `Y = X+1`
2	`fence(X)`	2 `fence(Y)`

```
assert(X≠Y)
```

Fig. 10. unsoundness example.

	(a) Concrete			(b) Predicate Abstraction		
	Thread 0 (X,Y)	**Thread 1 (X,Y)**	**Global (X,Y)**	**Thread 0** (X=Y,X=1,Y=1,X=0,Y=0)	**Thread 1** (X=Y,X=1,Y=1,X=0,Y=0)	**Global** (X=Y,X=1,Y=1,X=0,Y=0)
	(0,0)	(0,0)	(0,0)	(T, F, F, T, T)	(T, F, F, T, T)	(T, F, F, T, T)
T0: X = Y+1	(1,0)	(0,0)	(0,0)	(F, T, F, F, T)	(T, F, F, T, T)	(T, F, F, T, T)
T1: Y = X+1	(1,0)	(0,1)	(0,0)	(F, T, F, F, T)	(F, F, T, T, F)	(T, F, F, T, T)
T0: flush(X)	(1,0)	(1,1)	(1,0)	(**F**, **T**, F, F, T)	(F, T, T, F, F)	(**F**, **T**, F, F, T)
T1: flush(Y)	(1,1)	(1,1)	(1,1)	(F, T, F, F, T)	(**F**, T, **T**, F, F)	(**F**, T, **T**, F,F)

Fig. 11. (a) an error trace for Fig. 10 under PSO. Thread i are values observed by thread i, Global are the values in global memory. (b) Values of predicates in a boolean program corresponding to the program of Fig. 10 under PSO semantics.

To capture the assertion breach, we need to keep track of the relationship between X and Y. Consider the predicates:

$$P_1 : X = Y, P_2 : X = 1, P_3 : Y = 1, P_4 : X = 0, P_5 : Y = 0$$

Each P_i has been assigned a boolean variable B_i (with a buffer for each thread) where the effect of a thread writing to X or Y will write to a local buffer of $X = Y$ of that thread, and the effect of flushing X or Y will also flush the buffer associated with $X = Y$. Unfortunately this approach is insufficient. The problem is shown in Fig. 11(b). When thread 0 updates X to 1, and thread 1 updates Y to 1, the predicate $X = Y$ will be updated to *false* in both (denoted as F) and stored in the store buffer of each thread (since neither of the two threads can see the update of the other). When the value of X is flushed from the buffer of thread 0, our setting flushes the value of the predicate $X = Y$ (F) to main memory. Similarly, when Y is flushed in thread 1, the value of $X = Y$ (F) is flushed. The main memory state after these two flushes is inconsistent, as it has $X = 1$ set to T, $Y = 1$ set to T and $X = Y$ set to F.

A.4 Reduction TSO to SC

For x86 TSO, the store buffer is $B \in Thread \rightarrow (Gvar \times D)^*$. That is a sequence (store buffer) of (global variable, value) pairs per thread. The semantics of the language

Semantics 2 Operational semantics under TSO.

$$\frac{pc = l \quad {}^{l}r = X \quad \forall (Y,d) \in B.Y \neq X \quad G(X) = v \quad enabled(tid)}{L'(r) = v \quad pc' = n(pc)} \text{(LOAD-G)}$$

$$\frac{pc = l \quad {}^{l}r = X \quad B = b_1 \cdot (X,v) \cdot b_2 \quad \forall (Y,d) \in b_1.Y \neq X \quad enabled(tid)}{L'(r) = v \quad pc' = n(pc)} \text{(LOAD-B)}$$

$$\frac{pc = l \quad {}^{l}X = r \quad B = b \quad L(r) = v \quad enabled(tid)}{B' = b \cdot (X,v) \quad pc' = n(pc)} \text{(STORE)}$$

$$\frac{B = (X,v) \cdot b \quad enabled(tid)}{B' = b \quad G'(X) = v} \text{(FLUSH)}$$

$$\frac{pc = l \quad {}^{l}fence \quad B = \epsilon \quad enabled(tid)}{pc' = n(pc)} \text{(FENCE)}$$

$$\frac{pc = l \quad {}^{l}begin_atomic \quad in_atomic = \bot}{in_atomic = tid \quad pc' = n(pc)} \text{(BEGIN-ATOMIC)}$$

$$\frac{pc = l \quad {}^{l}end_atomic \quad in_atomic = tid}{in_atomic = \bot \quad pc' = n(pc)} \text{(END-ATOMIC)}$$

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

if $cnt_t = k$ **then**
 	
abort("overflow")
 	
$cnt_t = cnt_t + 1$
 	
if $cnt_t = 1$ **then**
  $lhs_{1_t} = X$
  $rhs_{1_t} = r$
...
if $cnt_t = k$ **then**
  $lhs_{k_t} = X$
  $rhs_{k_t} = r$
	if $cnt_t = 0$ **then goto** l_0
	...
	else if $cnt_t = k$ **then goto** l_k
	
	l_k:
	if $lhs_{k_t} = X$ **then**
	  $r = rhs_{k_t}$; **goto** l_{end}
	...
	l_1:
	if $lhs_1_t = X$ **then**
	  $r = rhs_1_t$; **goto** l_{end}
	
	l_0: $r = X$
	
	l_{end}:
		assume $(cnt_t = 0)$	**while** * **do**
			  if $cnt_t > 0$ **then**
			    if * **then**
			      ▷ for each $X \in Gvar$ generate:
			      if $lhs_{1_t} = X$ **then**
			        $X = rhs_{1_t}$
			      ▷ end of generation
			
			      if $cnt_t > 1$ **then**
			        $lhs_{1_t} = lhs_{2_t}$
			        $rhs_{1_t} = rhs_{2_t}$
			        ...
			        if $cnt_t = k$ **then**
			          $lhs_{(k-1)_t} = lhs_{k_t}$
			          $rhs_{(k-1)_t} = rhs_{k_t}$
			
			      $cnt_t = cnt_t - 1$

Fig. 12. TSO Translation Rules: each sequence of statements is atomic.

under the x86 TSO memory model are given n Semantics 2. Fig. 12 presents the translation of the four statements appearing in Semantics 2. In the translation, the newly generated sequence of statements is atomic.

Store to a global variable $[\![X = r]\!]^t_k$***:*** The store to a global variable X first checks if we are about to exceed the buffer bound k and if so, the program aborts. Otherwise, the counter is increased. The rest of the logic checks the value of the counter and updates the corresponding local variable. The global variable X is not updated and only local variables are involved.

Load from a global variable $[\![r = X]\!]^t_k$***:*** 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., $cnt_t = 0$), the load is performed directly from the global store. We do not need to check whether the buffer limit k is exceeded as that is ensured by the global store.

Fence statement $[\![fence]\!]^t_k$***:*** The fence waits for the buffer to be empty before executing.

Flush action $[\![flush]\!]^t_k$***:*** The flush action is translated into a loop with a non-deterministic exit condition (we use $*$). New statements are introduced for each global variable X. If the buffer counter is positive, (cnt_t is equal to some i where $1 \leq i < k$) and the top variable in the buffer (lhs_{i_t}) is X then it non-deterministically decides whether to update the global variable X or to continue the iteration. If it has decided to update X, the earliest write (i.e. rhs_{i_t}) is stored in X. The contents of the local variables are then updated by shifting: the content of each x_{j_t} is taken from the content of the successor $x_{(j+1)_t}$ where $1 \leq j < k$. Finally, the buffer count is decremented. The composite statement inside the while loop is generated for each global variable. To ensure a faithful translation of the flush action from Semantics 2, the whole newly generated statement is placed after *each* statement of the resulting program.

B Extrapolating Predicates: SC to TSO

In this section, we elaborate on how the function $preds_{tso}$ = EXTRAPOLATETSO($preds_{sc}$) operates. Fig. 13 shows a simple example in which a single thread stores two values into a shared variable X and then loads the value of X into $l1$. To successfully verify this program, the abstraction we use must be precise enough to capture intra-thread coherence.

Thread 1:

```
1  X=0;
2  X=1;
3  l1=X;
4  fence;
5  assert (X = l1);
```

Fig. 13. Intra-thread coherence example

B.1 Generic Predicates

If we use a buffer with a max size of 1, our translation adds the predicates: $cnt_t1 = 0, cnt_t1 = 1$ indicating the last location of the buffer which has been written to by thread 1, but not yet flushed. Having knowledge of the buffer structure, we need predicates about what is the actual content. To achieve that we first enumerate the possible shared variables in the system (e.g X as 0 and Y as 1) the actual order is of no significance and serves only as a means to reason about variables and not

just about their values. We add predicates of the form $(lhs_1_t1 = 0), (lhs_1_t1 = 1)$ - that track what variable is in buffer at location 1.

These predicates serve multiple purposes: (i) track the store buffers size; (ii) provide knowledge during `store` and `load` operations on where to write/read the value of X. (iii) preserve *Intra-thread coherence* in the abstraction.

Another predicate we generate is: $overflow = 0$. This predicate supplements the previously described predicates, giving indication of when the number of subsequent stores to a shared variable X, without a fence or a flush in between these stores, exceeds the limit k of our abstraction. This is crucial to ensure soundness of the abstraction.

The general extrapolation rule, which is independent of the verified program and of the input SC predicates $preds_{sc}$, is:

Rule 8 *For a buffer size bound k, add the following predicates to $preds_{tso}$:*

- $\{cnt_\mathbf{T} = i \mid 1 \leq i \leq k, \mathbf{T} \in Thread\}$
- $\{lhs_i_\mathbf{T} = s \mid 1 \leq i \leq k, 0 \leq s < \|V\|, \mathbf{T} \in Thread\}$
- $overflow = 0$

B.2 Extrapolating from $preds_{sc}$

We now describe how to extrapolate from the predicates in the set $preds_{sc}$ in order to compute new predicates that become part of $preds_{tso}$. The rule below ensures that the SC executions of the new program can be verified.

Rule 9 *Update the set $preds_{tso}$ to contain the set $preds_{sc}$*

Next, we would like properties on the values of a shared variable X captured by predicates in $preds_{sc}$ to also be captured for the buffered values of X. For example if $preds_{sc}$ contains $X = 0$, we add the predicate $rhs_1_t1 = 0$ for a buffer location that could hold X for thread T_1. This can be seen in the example of Fig. 13 where we need to track that the buffered value of X is 0 at line 1. We summarize these observations in the following rule:

Rule 10 *Update $preds_{tso}$ to contain the set:*

- $\bigcup_{p_{sc} \in preds_{sc}} lift(p_{sc})$

Here, $lift(p_{sc})$ generates from each SC predicate a set of TSO predicates where the original variables are replaced with buffered versions of the variables. (for each buffered version of a variable and their combination)

In addition to the above rules, adding a predicate $rhs_1_t1 = X$ ensures that the shared value of X and the buffered value of X are in agreement (when the predicate is set to $true$). This reduces the required state-space and enables faster conversion of model checking. Following similar reasoning, the predicate $rhs_1_t1 = rhs_2_t1$ is also added.

Rule 11 *For $\mathbf{V} \in Gvar$, $\mathbf{T} \in Thread$ and k the buffer bound, update $preds_{tso}$ to contain the sets:*

– $\{rhs_(i-1)_\mathbf{T} = rhs_i_\mathbf{T} \mid \; 2 \leq i \leq k\}$
– $\{rhs_i_\mathbf{T} = \mathbf{V} \mid 1 \leq i \leq k\}$

The above rules add both: generic predicates that are independent of $preds_{sc}$ as well as predicates that are extrapolated from $preds_{sc}$. But these rules may sometimes generate a larger set of predicates than necessary for successful verification. We now describe several optimizations that substantially reduce that number.

Rule 12 ***Read-only shared variables*** *If a thread never writes to a shared variable X do not extrapolate the SC predicates referencing X to their TSO counterparts for that thread.*

Rule 13 ***Predicates referencing a shared variable more than once*** *Replace all occurrences of the shared variable with the same buffered location.*

Next, for $Y2 \leq Y1$, we do not necessarily need to generate the predicate $rhs_1_t2 \leq rhs_1_t1$.

Rule 14 ***Predicates referencing different shared variables*** *For a predicate referencing more than one shared variable, if it can be guaranteed that a fence will be executed between every two shared location writes, restrict to generating predicates that relate to one buffered location at most.*

Additionally we note that some duplication in generation of predicates occur in the described procedure. For instance for the two predicates $X == 0$,$Y == 0$ they both will cause the same rhs_1_t1 TSO predicate to be generate.

C Proofs

We will prove the soundness of the translations introduced previously (Section A.4, Section 3).

For clarity we will denote next $[\![]\!]_k^{PSO}$ where we refer to $[\![]\!]_k$ as constructed following SC to PSO translation, and $[\![]\!]_k^{TSO}$ where SC to TSO translation was applied. The notation $[\![]\!]_k$ will still be used if the intention can be deduced from context.

Theorem 2 (Soundness of Translation). *For a given program, P and a safety specification S, if $P \not\models_{pso} S$ then there exists a $k \in \mathbb{N}$ such that $[\![P]\!]_k^{PSO} \not\models_{sc} S$. And if $P \not\models_{tso} S$ then there exists a $k \in \mathbb{N}$ such that $[\![P]\!]_k^{TSO} \not\models_{sc} S$.*

Lemma 1. *Assume $P \not\models_{pso} S(P \not\models_{tso} S)$, and let π be a trace witnessing it. There exists a $k \in \mathbb{N}$ such that at any point σ in π, for each global variable X, and every process T_i the buffer of X at T_i is filled with (strictly) less than k values (for TSO the buffer of T_i is filled with (strictly) less than k values at each point).*

Proof. The proof follows the same lines both for x86 TSO and PSO memory models.

Given that $P \not\models_{pso} S(P \not\models_{tso} S)$, let π be a trace in P witnessing it, i.e. $\pi \not\models_{pso} S$ ($\pi \not\models_{tso} S$), we will show $[\![\pi]\!]_k^{PSO} \not\models S$ (and accordingly $[\![\pi]\!]_k^{TSO} \not\models S$).

Since S is a safety property, π is finite. In addition, each transition can add only one entry to any of the local buffers - this implies Lemma 1.

We denote the following mapping from a state of P to a state of $[\![P]\!]_k$ under PSO: For each global variable X, a buffer entry h at process T_i will be mapped to x_{h_i} as described before for $h < k$ and to x_{k_i} for $h \geq k$.

Under TSO: For a process T_i, each buffer entry h will be mapped to rhs_{h_i} and lhs_{h_i} as described before for $h < k$ and to rhs_{k_i},lhs_{k_i} for $h \geq k$. Concretely - if at a TSO state that buffer entry holds a value $val1$, of a variable X that has an index $indx1$, under $[\![]\!]_k^{TSO}$ it will be mapped to $rhs_{h_i} = v$ and $lhs_{h_i} = indx1$ for $h < k$ and to $rhs_{k_i} = 0, lhs_{k_i} = 0$ for $h \geq k$. The mapping of global and local variables shall be the identity and the mapping of the instructions(transitions) given by the translation previously described.

The described mapping (that shall be denoted Ψ) is a Simulation from a subset of all possible execution of P, in which the buffer size does not exceed the limit of k for any global variable, to the sequentially consistent setting defined by our translation.

We will show this first for PSO and then for TSO.

1. We defined for each state its mapping.
2. Let σ, σ' be two states, where none of the buffers exceeds the limit of k, such that σ' is reachable from σ by the execution of an instruction S of T_i. We shall look at $\Psi(\sigma)$ and prove that $\Psi(\sigma')$ is his successor under $[\![S]\!]_k^i$.
   - **For** $S = (X = r)$ - From the constraint on σ' we can deduce that at σ the buffer of T_i for X, is of some size h such that $h + 1 < k$ (otherwise the buffer will exceed it at σ'). So if at $\Psi(\sigma)$ we had $x_{cnt_i} = h < k - 1$, $r = val1$ for some $val1$, than at $\Psi(\sigma')$, we will have $x_{cnt_i} = h + 1 < k$ and $x_{(h+1)_i} = val1$. Therefore $[\![S]\!]_k^i\,_{(\Psi(\sigma))} = \Psi(\sigma')$.
   - **For** $S = (r = X)$ - The T_i buffer of X at σ and σ' is either empty or some value $h, h < k$. Following this instruction r will receive at σ' a value $val1$ which resided previously either at location h of the buffer or at the global variable X. Either $x_{cnt_i} = h \leq k$, $h > 0$ and $x_{h_i} = val1$ at $\Psi(\sigma)$ or $h = 0$, and $X = val1$. For both cases we can safely deduce $r = val1$ both at σ' and at $\Psi(\sigma')$ and so we will have $[\![S]\!]_k^i\,_{(\Psi(\sigma))} = \Psi(\sigma')$
   - **For** $S = (fence)$ and **For** $S = (flush)$ - Similar reasoning apply.

Similar reasoning applies for TSO:

1. We defined for each state its mapping.
2. Let σ, σ' be two states, where none of the buffers exceeds the limit of k, such that σ' is reachable from σ by the execution of an instruction S of T_i. We shall look at $\Psi(\sigma)$ and prove that $\Psi(\sigma')$ is his successor under $[\![S]\!]_k^i$.
   - **For** $S = (X = r)$ where X is of index $indx1$- From the constraint on σ' we can deduce that at σ the buffer of T_i is of some size h such that $h + 1 < k$ (otherwise the buffer will exceed it at σ'). So if at $\Psi(\sigma)$ we had $cnt_i = h < k, r = val1$ for some $val1$, then at $\Psi(\sigma'), cnt_i = h + 1 \leq k, rhs_{(h+1)_i} = val1$ and $lhs_{(h+1)_i} = indx1$. Therefore $[\![S]\!]_k^i\,_{(\Psi(\sigma))} = \Psi(\sigma')$.

- **For** $S = (r = X)$ where X is of index $indx1$ - The T_i buffer of X at σ and σ' is either empty or some value $h, h < k$. Following this instruction r will receive at σ' a value $val1$ which resided previously either at location h of the buffer or at the global variable X. Either $cnt_i = h \leq k$, $h > 0$, $rhs_{h_i} = val1$ and $lhs_{h_i} = indx1$ at $\Psi(\sigma)$ or $h = 0$, and $X = val1$. For both cases we can safely deduce $r = val1$ both at σ' and at $\Psi(\sigma')$ and so we will have $[\![S]\!]^i_{k\ (\Psi(\sigma))} = \Psi(\sigma')$
- **For** $S = (fence)$ and **For** $S = (flush)$ - Similar reasoning apply.

The error trace π suffers no loss of precision under the mapping so $[\![\pi]\!]^{PSO}_k \not\models S$ ($[\![\pi]\!]^{TSO}_k \not\models S$), which means $[\![P]\!]_k \not\models_{SC} S$ both for PSO (and TSO).