Compiling Parameterized X86-TSO Concurrent Programs to Cubicle-$\mathcal{W}$

Sylvain Conchon[1,2], David Declerck[1,2], and Fatiha Zaïdi[1]

[1] LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay
[2] Inria, Université Paris-Saclay, F-91120 Palaiseau

Abstract. We present PMCx86, a compiler from x86 concurrent programs to Cubicle-$\mathcal{W}$, a model checker for parameterized weak memory array-based transition systems. Our tool handles x86 concurrent programs designed to be executed for an arbitrary number of threads and under the TSO weak memory model. The correctness of our approach relies on a simulation result to show that the translation preserves x86-TSO semantics. To show the effectiveness of our translation scheme, we prove the safety of parameterized critical primitives found in operating systems like mutexes and synchronization barriers. To our knowledge, this is the first approach to prove safety of such parameterized x86-TSO programs.

Keywords: Model Checking, MCMT, SMT, Weak Memory, x86, TSO

1 Introduction

Optimizations found in modern multiprocessors architectures affect the order in which memory operations from different threads may take place. For instance, on Intel x86 processors [20], each hardware thread has a write buffer in which it temporarily stores values before they reach the main memory. This allows the processor to execute the next instruction immediately but delays the store.

From an x86 programmer's point of view, the main drawback of this new memory model, called x86-TSO [24], is that most concurrent algorithms, designed under a global time (sequential consistency – SC) assumption [22], are incorrect on weaker semantics. However, while concurrent programming is known to be difficult, it is even harder to design correct programs when one has to deal with memory reordering.

This situation is further complicated by the fact that critical concurrent primitives found (for instance) in operating systems are usually designed to be executed for an arbitrary number of processes. Mutual exclusion algorithms or synchronization barriers are typical examples of such *parameterized* programs.

As a consequence, the design and verification of parameterized x86-TSO programs is a very hard challenge due to the state explosion problem caused by the combination of both *unbouded writing buffers* and *unbounded number of threads*.

Checking safety of programs running under a relaxed memory model has been shown to be a (non-primitive recursive-)hard problem [9, 11] and various verification techniques have been applied to handle it [3, 10, 12, 13, 16, 21, 23]. Among

those techniques, model checking of systems under weak memory assumption has been investigated and several tools have been implemented. The list of state-of-the-art model checkers for weak memory includes CBMC [7], MEMORAX [4] and TRENCHER [10].

Model checking has also been applied to parameterized systems for a long time ago [14, 8, 17] and automatic tools for the analysis of such systems exist. The list of state-of-the-art parametric model checkers includes MCMT [18], Undip [6], PFS [5] and Cubicle [15]. But until now, there is no model checker for reasoning about both weak memory and parameterized models, except Cubicle which has been extended recently to a new version, Cubicle-$\mathcal{W}$ [1], to verify parameterized array-based systems with weak memories.

In this paper, we present PMCx86 [2], a compiler from x86 assembly language to Cubicle-$\mathcal{W}$. The main originality of PMCx86 is that it can handle x86 concurrent programs designed to be executed for an arbitrary number of threads and under the TSO weak memory model. Our contributions are as follows:

- A compilation scheme from x86 to array-based transition systems with weak memory assumptions
- A simulation result to show that our translation preserves the TSO semantics
- An end-to-end tool that allows the verification of real critical x86 primitives found in OS like mutex or synchronization barriers.

To our knowledge, this is the first framework to model check parameterized x86-TSO concurrent programs.

In the remainder, we present in Section 2 the syntax and semantics of Cubicle-$\mathcal{W}$. In Section 3, we present the x86-TSO fragment supported by our framework. Section 4 is about the translation to Cubicle-$\mathcal{W}$. Finally Section 5 exhibits the experiments and the obtained results and we conclude and give some lines for future work in Section 6.

2 Overview of Cubicle-$\mathcal{W}$

In this section, we present the syntax and semantics of Cubicle-$\mathcal{W}$'s input language. This language is the target of our compiler PMCx86.

To illustrate our presentation, we use the crafted example shown in Figure 1. A Cubicle-$\mathcal{W}$ input file starts with enumerated type declarations (`type` keyword), followed by variables declarations. Thread-local (*i.e.* non shared) variables are declared as `proc`-indexed arrays. Those variables behave as *sequential consistent* (SC) memories. The `weak var` keyword is used to declare *shared* variables subject to weak memory effects. Similarly, shared weak arrays indexed by process identifiers are defined using `weak array` declarations. The initial states of the system are described by a (implicitly universally quantified) logical formula introduced by the `init` keyword. Similarly, the dangerous states are described by logical formulas introduced by the `unsafe` keyword and implicitly existentially quantified by process variables. Transitions are introduced by the `transition`

```
type loc = L1 | L2 | L3 | END

array PC[proc] : loc
weak var X : int
weak array A[proc] : int

init (p) {
  PC[p] = L1 && X = 0 && A[p] = 0 }

unsafe (p q) {
  PC[p] = End && PC[q] = End }
```

```
transition t1 (p)
requires { PC[p] = L1 }
{ p @ A[p] := 1; PC[p] := L2 }

transition t2 (p q)
requires { PC[p] = L2 &&
  fence(p) && p @ A[q] <> 0 }
{ PC[p] := L3 }

transition t3 (p)
requires { PC[p] = L3 }
{ p @ X := p @ X + 1; PC[p] := End }
```

Fig. 1. A crafted Cubicle-$\mathcal{W}$ example illustrating its syntactic features

keyword and are parameterized by existentially quantified process variables. Implicitly, the first parameter of each transition indicates which process performs the action. Each transition is composed of two parts: the *guard* and the *actions*. The *guard* is a logical formula that determines when the transition is enabled. The *actions* part is a set of updates on SC and weak variable. The *guard* evaluation and *actions* are performed *atomically*, *i.e.* no other transition can occur in between. In both parts, accesses to weak variables are performed using the `p @ X` notation, indicating that process p accesses the variable X. Cubicle-$\mathcal{W}$ imposes one restriction : All weak variable accesses in the same transition (guard and action) *must be* performed by the same process.

Cubicle-$\mathcal{W}$ simulates a write buffer semantics à la TSO for weak variables (or weak arrays). This means that each process has an associated FIFO-like write buffer, and when a transition performs writes to weak variables, all these writes are enqueued as a single update in the buffer. In a non-deterministic manner, an update may be dequeued from the buffer and committed to the weak variables. A process always knows the most recent value it wrote to a weak variables: when evaluating a read, a process first checks in its own buffer for the most recent write to the weak variable and returns the associated value, if any, otherwise it just returns the value from the variable itself. A transition *guard* may use a `fence(p)` predicate (as in transition `t2` for instance) to indicate that the transition may only be taken when process p's buffer is empty. When a transition contains both a read and a write (transition `t3`), it is given a *lock* semantics: it may be taken only when the buffer of the process performing the actions is empty (a `fence(p)` predicate is syntactically added to the transition *guard*), and the writes to weak variables bypass the buffer.

Formal semantics of Cubicle-$\mathcal{W}$

To make the semantics of Cubicle-$\mathcal{W}$ more formal, we give the pseudo-code of an interpreter for its transition systems in Algorithm 1. This interpreter makes use of data structures for buffers and some functions that we briefly describe here.

Buffers. A buffer (type `buffer`) is a queue containing updates. An update is made up of several writes, which associate a variable to a value. The operations on these buffers are:

- `is_empty`: determines if a buffer is empty
- `enqueue`: add an update at the head of the buffer
- `dequeue`: get and remove the update at the tail of the buffer
- `peek`: inspect every update from head to tail in the buffer until a given variable is found ; if it is, return the associated value, otherwise, return *None*

Auxiliary functions. The `upreg`(t) function returns the set of actions on local variables from a transition t. Similarly, the `upmem`(t) function returns the set of actions on weak variables. The `req`(t) function returns the whole transition *guard*. The `locked`(t) function determines if the transition has *lock* semantics. More importantly, the `eval`$(\mathcal{S}, e)$ function evaluates the expression e in state $\mathcal{S}$. It is trivial for most cases, except for reads and fences.

```
function eval(S, e) : begin
  match e with
    • i @ X →
      match peek(B[i], X) with
        • Some v → return v
        • None → return W[X]
      end
    • fence(i) → is_empty(B[i])
    • ... → ...
  end
end
```

The interpreter takes the form of an infinite loop that randomly chooses between executing a transition `t` ready to be triggered for some process arguments σ (*i.e.* `eval`$(\mathcal{S}, \texttt{req}(t)\sigma = true)$ or flushing a non-empty buffer $\mathcal{B}[i]$ of some process i. The execution of a transition first directly assigns local variables $R[i]$ in qthe (SC) memory. Then, it constructs an update value U with all pairs of (variable, value) corresponding to the weak assignments of `t`. If the transition has the *locked* semantics, this update value is enqueued in the buffer of the process which performs the action. Otherwise, its assignments are flushed in memory.

3 Supported x86-TSO fragment

We present in this section the subset of 32-bit x86 assembly instructions supported by our tool. In order to guide (and prove correct) our translation to Cubicle-$\mathcal{W}$, we also give an operational semantics of this fragment.

Algorithm 1: A Cubicle-$\mathcal{W}$ interpreter

Input: a number of processes n and a set of transitions τ
State: $\mathcal{S} = \{\ \mathcal{R}$: (`register` $\mapsto$ `value`) `map`
$\mathcal{W}$: (`variable` $\mapsto$ `value`) `map`
$\mathcal{B}$: (`proc` $\mapsto$ `buffer`) `map` $\}$
procedure `run`(n, τ) **: begin**
  while *true* **do**
    non-deterministically choose
      • *a transition t and a substitution σ s.t* `eval`$(\mathcal{S}$, `req`$(t)\sigma) = true \rightarrow$
        foreach *R[i] := e in* `upreg`(t) **do** $\mathcal{R}[R[i\sigma] \leftarrow$ `eval`$(\mathcal{S}, e\sigma)]$;
        $U := \emptyset$;
        foreach *$X := e$ in* `upmem`(t) **do**
          $U :=$ (X, `eval`$(\mathcal{S}, e\sigma)) + U$
        end foreach
        if `locked`$(t) =$ *false* **then**
          `enqueue`$(\mathcal{B}[i], U)$
        else
          foreach *(X, v) in U* **do** $\mathcal{W}[X \leftarrow v]$
        end if
      • *a process i s.t* `is_empty`$(\mathcal{B}[i]) =$ *false* $\rightarrow$
        let $U =$ `dequeue`$(\mathcal{B}[i])$ **in**
        foreach *(X, v) in U* **do** $\mathcal{W}[X \leftarrow v]$
    or exit if no choice possible
  end while
end

Input programs are written in a NASM-like syntax. The six general purpose registers `eax`, `ebx`, `ecx`, `edx`, `esi` and `edi` are available. Instruction operands may be registers, immediate data, and direct memory references of the form `[var]`. Memory accesses occur under the TSO weak memory semantics. The supported instructions are:

- Load/store: `mov`
- Arithmetic: `add`, `sub`, `inc`, `dec`
- Exchange: `xadd`, `xchg`
- Compare: `cmp`
- Jump: `jmp`, `jCC`
- Memory ordering: `mfence`, `lock` prefix (on `add`, `sub`, `inc`, `dec`, `xadd`, `xchg`)

In order to write (and translate) parametric programs, we allow data declarations to be decorated with an annotation `! as counter` which specifies a counter on the number of threads. These counters are still treated as regular integers on x86, but may only be manipulated by the `inc`, `dec`, `cmp` and `mov` instructions. Moreover, they will be translated differently in Cubicle-$\mathcal{W}$.

For the sake of simplicity, we only introduce in this section the most relevant aspects of our fragment, as shown in the grammar in Figure 2. The interested

reader may refer to Appendix A for a detailed grammar of the supported fragment.

integer, n		integer
register, r		register
variable, x		variable
label, l		instruction label (index in instruction array)
thread_id, tid		thread identifier
instruction, i	::=	
	| mov *r, n*	load a constant into a register
	| mov *r, x*	load a variable into a register
	| mov *x, n*	write a constant into a variable
	| mov *x, r*	write the contents of a register into a variable
	| add *x, r*	add the contents of a register to a variable
	| inc *x*	increment a variable by 1
	| cmp *r, r'*	compare the contents of two registers
	| cmp *x, n*	compare a variable to a constant
	| je *l*	branch if equal (ZF=1)
	| jne *l*	branch if different (ZF=0)
	| lock *i*	lock prefix to perform atomic RMW instructions
	| mfence	memory fence
action, a	::=	
	| *i*	the execution of an instruction
	| ϵ	flush of a buffer
thread, t	::=	
	| (*i array*)	array of instructions
program, p	::=	
	| (*tid* $\mapsto$ *t*) *map*	a map of threads identifiers to thread instructions

Fig. 2. Abstract syntax tree of x86 program

In our abstract syntax, a *thread* is described by an array of instructions and a *program* is just a map from thread identifiers to threads. A thread executes an *action* which is either an instrucion or a flush of its buffer.

Representation of x86-TSO states

In order to give the semantics of this fragment, we need to define an x86-TSO memory state S. Such state is composed of two parts: the set of thread's local states LS and the shared memory M. Each thread local state ls is composed of its *instruction pointer* `eip`, its *zero flag* `zf`, its set of *registers* Q, and its writing buffer B.

$S = (LS \times M)$	An x86-TSO machine state
$M = (var \mapsto int)\ map$	A memory: dictionary from variables to integers
$LS = (tid \mapsto ls)\ map$	The thread local states: dictionary from

	thread identifiers to their states
$ls = (\texttt{eip} \times \texttt{zf} \times Q \times B)$	A thread local state
$Q = (reg \mapsto int)\ map$	A thread's registers: dictionary from register names do integers
$B = (var \times int)\ queue$	A thread's TSO write buffer
$\texttt{eip} = int$	A thread's instruction pointer
$\texttt{zf} = int$	A thread's *zero flag*
var	The set of all variables
tid	The set of all thread identifiers
reg	The set of all register names

The `eip` register represents a thread's *instruction pointer*, *i.e.* the threads's current program point. For simplicity, we choose to represent it using an integer type. The `zf` is a boolean register, commonly used to store the result of the compare instruction, and more generally of any arithmetic instruction. It must be set to *true* if the result of the last instruction was 0, and *false* otherwise. We represent it using an integer, with the usual convention that 0 = *false* and 1 = *true*. Writing buffers are represented as queues containing pairs of variables and integers. Initially, the x86-TSO machine is in a state S_{init} such that all thread's `eip` are set to 0, all buffers are empty, and all registers and the shared memory are in an undetermined state.

Notations for manipulating TSO buffers

When describing the instruction semantics, we use the following notations to manipulate the buffers:

$x \in B$	True if at least one pair in B concerns variable x
$x \notin B$	True if no pair in B concerns variable x
$B = \varnothing$	True if B is empty
$B_1 +\!\!+ B_2$	Concatenation of B_1 and B_2
$(x, n) +\!\!+ B$	Prepend (x, n) to the head of B
$B +\!\!+ (x, n)$	Append (x, n) to the tail of B

Semantics of programs

The semantics of a program is defined by the smallest relation $\xrightarrow{tid:a}$ on x86-TSO machine states that satisfies the rule Scheduling below.

We define a scheduling function $\pi_p(S)$ which given a program p and a state S chooses the next action to be executed by a thread.

$$\frac{\pi_p(LS, M) = tid : a \quad LS(tid) = ls \quad (ls, M) \xrightarrow{a} (ls', M')}{(LS, M) \xrightarrow{tid:a} (LS[t \mapsto ls'], M')}\ \textsc{Scheduling}$$

Semantics of instructions

The semantics of instructions is given by the smallest relation $\xrightarrow{i}$ that satisfies the rule below.

There are as many rules as required to cover the different combination of operand kinds (constant, register, memory). Here, for the sake of readability, we only present some rules that are TSO specific and explain the main principles of that relaxed memory semantics.

The rule MovVarCst assigns a shared variable x with a constant n. As assignments are delayed in TSO, a new pairs (x, n) is enqueued in buffer B.

$$\frac{a = \mathsf{mov}\ x, n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, zf, Q, (x, n) +\!\!+ B), M)}\ \textsc{MovVarCst}$$

The next two rules illustrate the TSO semantics of an instruction `mov` r, x that assigns to a register r the contents of a shared memory x. When the thread's buffer does not have a write on x, MovRegVarM applies and the value of x is directly read in memory.

$$\frac{a = \mathsf{mov}\ r, x \quad x \notin B \quad M(x) = n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, zf, Q[r \mapsto n], B), M)}\ \textsc{MovRegVarM}$$

On the contrary, when the thread's buffer contains a pair (x, n), rule MovRegVarB looks for the value of the most recent assignment to x in B.

$$\frac{a = \mathsf{mov}\ r, x \quad B = B_1 +\!\!+ (x, n) +\!\!+ B_2 \quad x \notin B_1}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, zf, Q[r \mapsto n], B), M)}\ \textsc{MovRegVarB}$$

The semantics of non-atomic read-modify-write instructions like `add` is still given by a single rule. Indeed, since the write is buffered, it has the same effect as splitting the read and the write in two rules.

$$\frac{a = \mathsf{add}\ x, r \quad x \notin B \quad M(x) + Q(r) = n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, \mathsf{iszero}(n), Q, (x, n) +\!\!+ B), M)}\ \textsc{AddVarMReg}$$

$$\frac{a = \mathsf{add}\ x, r \quad B = B_1 +\!\!+ (x, m) +\!\!+ B_2 \quad x \notin B_1 \quad m + Q(r) = n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, \mathsf{iszero}(n), Q, (x, n) +\!\!+ B), M)}\ \textsc{AddVarBReg}$$

When the lock prefix is used on a read-modify-write instruction, we simply require the thread's buffer to be empty, and directly write to memory.

$$\frac{a = \mathsf{lock\ add}\ x, r \quad x \notin B \quad M(x) + Q(r) = n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, \mathsf{iszero}(n), Q, B), M[x \mapsto n])}\ \textsc{LockAddVarReg}$$

Last, the rule MFence describes the effect of a memory fence which enforces a thread buffer to be flushed.

$$\frac{a = \mathsf{mfence} \quad B = \varnothing}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, zf, Q, B), M)} \text{ MFence}$$

Buffer / Memory synchronization

Buffers can flush their oldest writes to memory in an asynchronous manner. We express this using a rule that only involves the state of buffers, without involving the *eip* registers.

$$\frac{a = \epsilon \quad B = B_1 +\!\!+ (x, n)}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip, zf, Q, B_1), M[x \mapsto n])} \text{ WriteMem}$$

4 Translation to Cubicle-$\mathcal{W}$

We represent x86-TSO states in Cubicle-$\mathcal{W}$ by a set of variables corresponding to the shared variables and local states of each thread. Local registers are encoded as elements of an array indexed by process identifiers. The type of the array depends on the kind of values carried by the registers.

Instruction pointers. They are represented by a `PC` array. Program points are given an enumerated type *loc*, whose elements are of the form L_0, ... L_1. The number of these elements can be determined statically at compile time: it depends on the length of the longest instruction sequence.

Zero flags. They are represented by a `ZF` array of type `int`. We use the convention that $n = 0 \equiv true$ and $n \neq 0 \equiv false$. Note that this is the opposite of x86: this allows to compute this flag more easily, as we simply set it to the result of the last arithmetic operation. This allows to reduce the number of transitions, as we do not have to make any further operation to compute it.

Shared variables. Each shared variable `X` gives rise to an `weak X : int` declaration. Counters (see below) are mapped to `weak` arrays of type `bool`.

Translation of x86-TSO instructions

For the sake of readability, we only give the translation of the subset of instructions given in the previous section.

We define a compilation function $\mathcal{C}$ that takes as input a thread identifier, an instruction, and the instruction position in the array (this is equivalent to the *instruction pointer*). It returns a set of Cubicle-$\mathcal{W}$ transitions that simulates the instruction.

The first rule TMovVarCst explains how to translate the write of a constant into a shared variable. It simply amounts to use Cubicle-$\mathcal{W}$'s write instruction on

weak variables. This instruction imposes to prefix the operation with the thread identifier which performs the assignment.

$\mathcal{C}(\texttt{t}\ ;\ \mathsf{mov}\ x, n\ ;\ i) =$ TMovVarCst

```
transition mov_var_cst_i(t)
requires { PC[t] = L_i }
{ t @ X := n; PC[t] = L_{i+1} }
```

The next rule is the opposite operation: the read of a variable into a register. To achieve it, we only rely on the *read* operation on shared variables. Similarly to write instructions, reads must be prefixed with a thread identifier.

$\mathcal{C}(t\ ;\ \mathsf{mov}\ r, x\ ;\ i) =$ TMovRegVar

```
transition mov_reg_var_i(t)
requires { PC[t] = L_i }
{ R[t] := t @ X; PC[t] = L_{i+1} }
```

Adding the contents of a register to a variable is a read-modify-write operation. As Cubicle-$\mathcal{W}$ makes everything inside a transition atomic, we need two transitions to translate this operation. The first one, TAddVarReg1, reads the shared variable `X`, sums it with the local register and stores the result into a temporary register `T[t]`. The second rule, TAddVarReg2, stores the contents of this temporary register into the variable `X` and updates the *zero flag* accordingly.

$\mathcal{C}(t\ ;\ \mathsf{add}\ x, r\ ;\ i) =$ TAddVarReg1

```
transition add_var_reg_1_i(t)
requires { PC[t] = L_i }
{ T[t] := t @ X + R[t]; PC[t] = L_{xi} }
```

TAddVarReg2

```
transition add_var_reg_2_i(t)
requires { PC[t] = L_{xi} }
{ t @ X := R[t]; ZF[t] := T[t];
  PC[t] = L_{i+1} }
```

Translating the atomic counterpart of this operation is very simple, since Cubicle-$\mathcal{W}$ transitions are atomic. We just need a single transition, as given by rule TLockAddVarReg.

$\mathcal{C}(t\ ;\ \mathsf{add}\ x, r\ ;\ i) =$ TLockAddVarReg

```
transition lockadd_var_reg_i(t)
requires { PC[t] = L_i }
{ t @ X := t @ X + R[t];
  ZF[t] := t @ X + R[t];
  PC[t] = L_{i+1} }
```

The translation of a memory fence simply relies on the *fence* predicate of Cubicle-$\mathcal{W}$ to express that the transition may only be taken if the thread's buffer is empty.

$\mathcal{C}(t\ ;\ \mathsf{mfence}\ ;\ i) =$ TMFence

```
transition mfence_i(t)
requires { PC[t] = L_i && fence(t) }
{ PC[t] = L_{i+1} }
```

Translation of operations on counters

Operations on counters are restricted and translated differently. When `X` is a variable declared with a `! as counter` annotation, our tool only supports the following operations:

`mov X, 0`	reset
`inc X`	incrementation
`cmp X,` N	comparison to N
`cmp X, 0`	

where N is an abstract value represented the (parameterized) number of threads.

At first sight, it would be tempting to translate counters directly as variables of type `int`. However, this solution makes it impossible to compare a counter with N as Cubicle does not explicitly provide this constant. To solve this issue, we represent counters by weak arrays of Booleans indexed by processes. Each operation is then encoded in a unary numeral system. In the rest of this section, we only describe the first three ones.

Reset. To reset a counter, we just need to apply the transition given by the rule below which writes the value `False` in all the array cells.

$\mathcal{C}(t\ ;\ \mathsf{mov}\ x, n\ ;\ i) =$ TMovCnt0

```
transition mov_cnt0_i(t)
requires { PC[t] = L_i }
{ t @ X[k] := case | _ : False;
  PC[t] = L_{i+1} }
```

Incrementation. As for the incrementation of shared variables, a counter incrementation has to be performed in two steps. This first one for reading the contents of the variable and the second one adding one and assigning the new value. In our unary numeral system, adding one to a variable amounts to switch an array cell from `False` to `True`. The goal of the first transition is thus to find a cell equal to `False` and the second rule performs the assignment to `True`. The rules are duplicated as we may either switch the cell belonging to the running thread or to another thread.

$\mathcal{C}(t\ ;\ \mathsf{inc}\ x\ ;\ i) =$

```
transition inc_cntS_1_i(t)                                  TIncCntS1
requires { PC[t] = L_i && t @ X[t] = False }
{ PC[t] = L_{xi} }

transition inc_cntS_2_i(t)                                  TIncCntS2
requires { PC[t] = L_{xi} }
{ t @ X[t] := True; ZF[t] := 1;
  PC[t] = L_{i+1} }

transition inc_cnt0_1_i(t o)                                TIncCntO1
requires { PC[t] = L_i  && t @ X[o] = False}
{ PC[t] = L_{yi}; TP[t] = o }

transition inc_cnt0_2_i(t o)                                TIncCntO2
requires { PC[t] = L_{yi} && TP[t] = o}
{ t @ X[o] := True; ZF[t] := 1;
  PC[t] = L_{i+1} }
```

Comparison. We design three transitions for comparing a counter with the (parametric) number N of threads. To check if a counter is equal to N, we just check whether all cell of the array are `True`, using a universally quantified process variable. If it is the case, then the counter has reached the total number of threads, and the *zero flag* is set to 0. To check the opposite, we check if there exists a cell with the value `False`. In that case, the counter has not reached the total number of threads yet, so the *zero flag* is set to 1. Note that we need two transitions to achieve this: one to compare the cell owned by the executing thread, and another to compare any other cell.

$\mathcal{C}(t\ ;\ \texttt{cmp}\ x, N\ ;\ i) =$ TCmpCntEqN

```
transition cmp_cnt_eqN_i(t)
requires { PC[t] = L_i
   && t @ X[t] = True
   && forall_other o.
      t @ X[o] = True }
{ ZF[t] = 0; PC[t] = L_xi }
```

TCmpCntSNeqN

```
transition cmp_cntS_NeqN_i(t)
requires { PC[t] = L_xi
   && t @ X[t] = False }
{ ZF[t] := 1; PC[t] = L_i+1 }
```

TCmpCntONeqN

```
transition cmp_cntO_NeqN_i(t o)
requires { PC[t] = L_xi
   && t @ X[o] = False }
{ ZF[t] := 1; PC[t] = L_i+1 }
```

Translation of programs

In order to compile all instructions of a thread, we define a compilation function $\mathcal{C}_t$ that takes as input a thread identifier and an instruction array. This function returns the set of Cubicle-$\mathcal{W}$ transitions corresponding to the translation of every instruction in the array.

$$\mathcal{C}_t(tid\ ;\ t) = \bigcup_{i=1}^{|t|} \mathcal{C}(tid\ ;\ t(i)\ ;\ i)$$

Similarly, we define a compilation function $\mathcal{C}_p$ that takes as input an x86 program and returns the set of transitions corresponding to the translation of every instruction in every thread.

$$\mathcal{C}_p(p) = \bigcup_{tid\ \in\ dom(p)} \mathcal{C}_t(tid\ ;\ p(tid))$$

Correctness

In order to prove the correctness of our approach, we demonstrate a simulation lemma between x86 programs and weak array-based transition systems obtained by translation.

Let $S = (LS \times M)$ be an x86-TSO machine state. Translating S to a Cubicle-$\mathcal{W}$ state A is straighforward, except for the memory map M and the contents of each thread buffer. For that, we define a predicate $\mathcal{T}(S, A)$ on Cubicle-$\mathcal{W}$ states such that $\mathcal{T}(S, A)$ is true if and only if:

- Local thread registers, *eip* and flags in LS contain the same values as their array-based representation
- For each local buffer B of a thread `tid` and for all shared variable `X`
  if ($\texttt{X} \notin B$ and $M(\texttt{X}) = v$) or ($B = B_1 +\!\!+ (\texttt{X}, v) +\!\!+ B_2$ and $\texttt{X} \notin B_1$) then
  - if `X` is a counter, then `tid @ X[k]` = `True` is true for v thread identifiers in A
  - otherwise, `tid @ X` $= v$ is true in A

Lemma 1 (Simulation). *For all program p and state S, if $S \xrightarrow{tid:a} S'$ then their exists a Cubicle-$\mathcal{W}$ state A such that $\mathcal{T}(S, A)$ is true and $\mathcal{C}_p(p)$ can move from A to A' and $\mathcal{T}(S', A')$ is true as well.*

Proof. By a simple inspection of each transition rule of x86 instructions. See Appendix B for more details.

Theorem 1. *Given a program p, if Cubicle-$\mathcal{W}$ returns* safe *on $\mathcal{C}_p(p)$ then p cannot reach an unsafe state (as described in the section* `unsafe_prop` *of p).*

Proof. By induction on the length of the reduction $p \xrightarrow{tid:a}{}^{+} \bot$ and by case on each step (using simulation lemma).

5 Experiments

We used our framework to translate and check the correctness of several mutual exclusion algorithms, as well as a sense reversing barrier algorithm. In this section, we only describe two of them. The source code and Cubicle-$\mathcal{W}$ translations of all the examples can be found on the tool page [2].

Figure 3 is a spinlock implementation found in the Linux kernel (version 2.6.24.7), and is an example used in [24]. It requires a single shared variable `Lock`, initialized to 1, and the use of the *lock dec* instruction. The lock prefix is required to make this algorithm correct. To enter the critical section, a thread t has to atomically decrement the contents of the `Lock` variable. It then checks the result of the operation, using a conditional branch instruction: if the result is not negative, it means that `Lock` was 1 before the decrement, so t performs the branch to enter the critical section. If the result is negative, it means that `Lock`

was 0 or less before the decrement, so another thread is already in the critical section : t enters a spinlock, waiting for `Lock` to be 1 before retrying to enter the critical section. To release the lock, the thread in critical section simply sets back `Lock` to 1.

```
begin shared_data
    Lock  dd  1
end shared_data

begin unsafe_prop
    eip[$t1] = cs &&
    eip[$t2] = cs
end unsafe_prop

begin init_code
    start_threads
end init_code
```

```
begin thread_code
acquire: lock dec dword [Lock]
         jns cs
spin:    cmp dword [Lock], 0
         jle spin
         jmp acquire
cs:      ; critical section
exit:    mov dword [Lock], 1
         jmp acquire
end thread_code
```

Fig. 3. A Linux Spinlock implementation

Our next example shown in Figure 4 is a Sense Reversing Barrier[19]. It allows a number of threads to synchronize their execution flow at a specific program point. It requires a process counter `count` and a boolean variable `sense` that gives the sense of the barrier. It locally uses the `esi` register to track the current value of the `sense` variable. Initially, `count` is set to N, and `sense` and `esi` are to 0. Threads start by reversing `esi`. Then, each thread atomically decrements the `count` variable. If, as a result of this operation, the `count` is not 0, then the thread enters a spinlock that waits for `sense` to be equal to `esi` (that is, for the barrier sense to be changed). If however the `count` reaches 0, then the thread resets `count` to N and copies `esi` into `sense`, which in effect reverses the sense of the barrier. At this point, threads that were waiting at the spinlock are released.

The results of our experiments are given in the table below. As a measure of the problem's complexity, we give the number of Registers, Weak variable and Transitions of the corresponding Cubicle-$\mathcal{W}$ program. The CE Length column gives the length of the counter-example, where applicable. It is the smallest number of transitions that lead to a state that violates the safety property. The Time column is the total time to prove the safety property (or to exhibit a counter-example). We considered both correct (S) and incorrect (US) versions of program. Incorrect versions are obtained by removing the lock prefixes.

```
begin shared_data
    sense  dd  0
    count  dd  N ! as counter
end shared_data

begin unsafe_prop
    eip[$t1] = entr &&
    eip[$t2] = end
end unsafe_prop

begin init_code
    start_threads
end init_code
```

```
begin thread_code
      mov esi, 0 ; esi = local sense
entr: not esi
      lock dec dword [count]
      jne spin
last: mov dword [count], N
      mov dword [sense], esi
      jmp end
spin: cmp dword [sense], esi
      jne spin
end:  nop
end thread_code
```

Fig. 4. A Sense Reversing Barrier algorithm in x86

Case study	Regs.	Weak Vars.	Trans.	CE Length	Time
naive mutex (dlock.) (US)	3	2	11	12	0,38s
naive mutex (no dlock.) (US)	3	2	14	12	0,38s
mutex w/ *xchg* (US)	4	1	8	10	0,07s
mutex w/ *xchg* (S)	3	1	7	-	0,08ss
mutex w/ *cmpxchg* (US)	4	1	10	10	0,12s
mutex w/ *cmpxchg* (S)	4	1	8	-	0,47s
Linux spinlock (US)	4	1	10	6	0,06s
Linux spinlock (S)	4	1	9	-	0,30s
sense barrier (sing. ent.) (S)	3	2	15	-	0,27s
sense barrier (mult. ent.) (S)	3	2	16	-	1m37s

6 Conclusion & Future Work

We have presented in this paper a compilation scheme from parameterized x86-TSO programs to weak array-based transitions systems in Cubicle-$\mathcal{W}$. The subset of the 32-bit x86 assembly instructions supported allows us to express critical concurrent primitives like mutexes and synchronization barriers. Experiments are promising. To our knowledge, this is the first tool for proving automatically the safety of parameterized x86-TSO programs.

An immediate line of future work is to enhance the subset of x86 that is supported according to new experiments that will be conducted. These adding should also be proved correct regarding the preservation of the TSO semantics. Another line of work will be to consider others memory models as a given input to the model checker and to make change in its reachability analysis algorithm accordingly.

References

1. Cubicle-W, `https://www.lri.fr/~declerck/cubiclew/`
2. PMCX86, `https://www.lri.fr/~declerck/pmcx86/`
3. Abdulla, P.A., Atig, M.F., Chen, Y., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: TACAS. pp. 204–219 (2012)
4. Abdulla, P.A., Atig, M.F., Chen, Y., Leonardsson, C., Rezine, A.: Memorax, a precise and sound tool for automatic fence insertion under TSO. In: TACAS (2013)
5. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers. In: TACAS. Springer (2007)
6. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: CAV. Springer (2007)
7. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. ESOP (2013)
8. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (May 1986)
9. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL. pp. 7–18 (2010)
10. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against tso. pp. 533–553. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013)
11. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: ICALP. pp. 428–440 (2011)
12. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV. pp. 107–120 (2008)
13. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: TACAS. pp. 11–25 (2011)
14. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about networks with many identical finite-state processes. PODC '86, ACM, New York, NY, USA (1986)
15. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel smt-based model checker for parameterized systems: Tool paper. In: CAV. pp. 718–724. CAV'12, Springer-Verlag, Berlin, Heidelberg (2012)
16. Dan, A.M., Meshman, Y., Vechev, M.T., Yahav, E.: Effective abstractions for verification under relaxed memory models. In: VMCAI. pp. 449–466 (2015)
17. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (Jul 1992)
18. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: IJCAR. pp. 22–29 (2010)
19. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
20. Intel Corporation: Intel 64 and IA-32 Architectures SDM (Dec 2016)
21. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI. pp. 187–198 (2011)
22. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. 28(9), 690–691 (Sep 1979)
23. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: TACAS. pp. 339–353 (2013)
24. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. CACM 53(7) (Jul 2010)

A Supported x86 fragment grammar

⟨*digit*⟩ ::= `0-9`
⟨*alpha*⟩ ::= `a-z A-Z`
⟨*ident*⟩ ::= (⟨*alpha*⟩ | `_`) (⟨*alpha*⟩ | `_` | ⟨*digit*⟩)⋆

⟨*integer*⟩ ::= ⟨*digit*⟩+
⟨*label*⟩ ::= ⟨*ident*⟩ `:`
⟨*thread*⟩ ::= `$`⟨*ident*⟩
⟨*reg*⟩ ::= `eax` | `ebx` | `ecx` | `edx` | `esi` | `edi`

⟨*program*⟩ ::= ⟨*shareddata*⟩?
⟨*threaddata*⟩?
⟨*threadcode*⟩
⟨*unsafeprop*⟩

⟨*shareddata*⟩ ::= `begin shared_data NL`
⟨*dline*⟩⋆
`end shared_data NL`

⟨*threaddata*⟩ ::= `begin thread_data NL`
⟨*dline*⟩⋆
`end thread_data NL`

⟨*threadcode*⟩ ::= `begin thread_code NL`
⟨*cline*⟩⋆
`end thread_code NL`

⟨*unsafeprop*⟩ ::= `begin unsafe_prop NL`
⟨*atom*⟩ (`&&` ⟨*atom*⟩)⋆
`end unsafe_prop NL`

⟨*dline*⟩ ::= ⟨*ident*⟩ `dd` ⟨*integer*⟩ ⟨*dannot*⟩? `NL`
⟨*cline*⟩ ::= ⟨*label*⟩⋆ ⟨*instr*⟩ `NL`

⟨*atom*⟩ ::= ⟨*term*⟩ ⟨*op*⟩ ⟨*term*⟩
⟨*op*⟩ ::= `=` | `<>` | `<` | `>` | `<=` | `>=`
⟨*term*⟩ ::= ⟨*treg*⟩ | ⟨*tlvar*⟩ | ⟨*tvar*⟩ | ⟨*ident*⟩
⟨*treg*⟩ ::= ⟨*reg*⟩ `[` ⟨*thread*⟩) `]`
⟨*tlvar*⟩ ::= ⟨*ident*⟩ `[` ⟨*thread*⟩) `]`
⟨*tvar*⟩ ::= ⟨*thread*⟩ `:` ⟨*ident*⟩ (`[` ⟨*thread*⟩) `]`)?

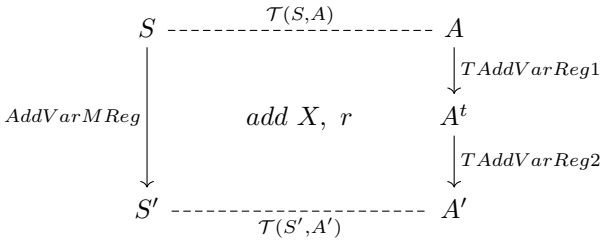
⟨*dannot*⟩ ::= `! as counter`

⟨*instr*⟩ ::= `inc` ⟨*oprm*⟩
| `dec` ⟨*oprm*⟩
| `not` ⟨*oprm*⟩
| `add` ⟨*oprm*⟩ `,` ⟨*oprmi*⟩
| `sub` ⟨*oprm*⟩ `,` ⟨*oprmi*⟩
| `xchg` ⟨*oprm*⟩ `,` ⟨*oprm*⟩
| `xadd` ⟨*oprm*⟩ `,` ⟨*opr*⟩
| `cmp` ⟨*oprm*⟩ `,` ⟨*oprmi*⟩
| `mov` ⟨*oprm*⟩ `,` ⟨*oprmi*⟩
| `jmp` ⟨*ident*⟩
| `jCC` ⟨*ident*⟩
| `nop`
| `mfence`
| `lock` ⟨*instr*⟩

⟨*opr*⟩ ::= ⟨*reg*⟩
⟨*oprm*⟩ ::= ⟨*reg*⟩ | ⟨*mem*⟩
⟨*oprmi*⟩ ::= ⟨*reg*⟩ | ⟨*mem*⟩ | ⟨*imm*⟩
⟨*mem*⟩ ::= `[` ⟨*ident*⟩ (`+` ⟨*thread*⟩)? `]`
⟨*imm*⟩ ::= ⟨*integer*⟩ | ⟨*ident*⟩

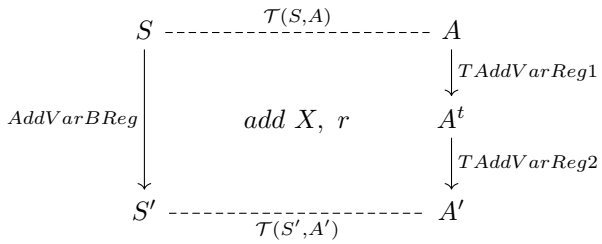
B Correctness

Add rule (from memory)



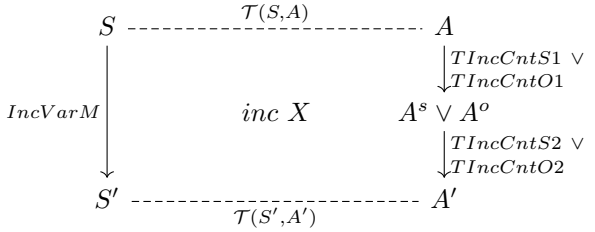
Let t be a thread and S an x86-TSO state of the form (LS, M) where $LS(t) = (eip, zf, Q, B)$ such that $\mathtt{X} \notin B$ and $M(\mathtt{X}) = n$. By rule AddVarMReg, the x86-TSO program can move to a state S' of the form (LS', M) where $LS'(t) = (eip', zf', Q, B')$ such that $B' = (\mathtt{X}, n + Q(r)) +\!\!+ B$ and $\forall t' \neq t. LS'(t') = LS(t')$. Let A be a state such that $\mathcal{T}(S, A)$ is *true*. In particular, the `PC[t]` register in A is equivalent to its counterpart in S, and `t @ X =` n is *true*. Then the rule TAddVarReg1 applies, and the Cubicle-$\mathcal{W}$ program reaches a state A^t in which `T[t] =` n `+ R[t]`. From this state, the rule TAddVarReg2 applies and performs the operation `t @ X := T[t]`, making the program reach a state A' such that `t @ X =` n `+ R[t]` is *true* and thus $\mathcal{T}(S', A')$ is *true*.

Add rule (from buffer)



Let t be a thread and S an x86-TSO state of the form (LS, M) where $LS(t) = (eip, zf, Q, B)$ such that $B = B_1 +\!\!+ (\mathtt{X}, n) +\!\!+ B_2$ and $\mathtt{X} \notin B_1$. By rule AddVarBReg, the x86-TSO program can move to a state S' of the form (LS', M) where $LS'(t) = (eip', zf', Q, B')$ such that $B' = (\mathtt{X}, n + Q(r)) +\!\!+ B$ and $\forall t' \neq t. LS'(t') = LS(t')$. Let A be a state such that $\mathcal{T}(S, A)$ is *true*. In particular, the `PC[t]` register in A is equivalent to its counterpart in S, and `t @ X =` n is *true*. Then the rule TAddVarReg1 applies, and the Cubicle-$\mathcal{W}$ program reaches a state A^t in which `T[t] =` n `+ R[t]`. From this state, the rule TAddVarReg2 applies and performs the operation `t @ X := T[t]`, making the program reach a state A' such that `t @ X =` n `+ R[t]` is *true* and thus $\mathcal{T}(S', A')$ is *true*.

Inc rule with counters (from memory)



Let t be a thread and S an x86-TSO state of the form (LS, M) where $LS(t) = (eip, zf, Q, B)$ such that $\mathtt{X} \notin B$ and $M(\mathtt{X}) = n$. By rule IncVarM, the x86-TSO program can move to a state S' of the form (LS', M) where $LS'(t) = (eip', zf', Q, B')$ such that $B' = (\mathtt{X}, n + 1) +\!\!+ B$ and $\forall t' \neq t. LS'(t') = LS(t')$. Let A be a state such that $\mathcal{T}(S, A)$ is *true*. In particular, the `PC[t]` register in A is equivalent to its counterpart in S, and as `X` is a counter, `t @ X[k] = True` is *true* for exactly n threads. From now on, two cases may apply :

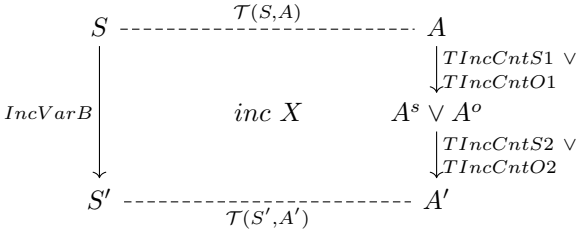
Case 1. `t @ X[t] = False` is *true*. Then the rule TIncCntS1 applies, and the Cubicle-$\mathcal{W}$ program reaches a state A^s. From this state, the rule TIncCntS2 applies and performs the operation `t @ X[t] := True`, making the program reach a state A' such that `t @ X[k] = True` is *true* for exactly $n + 1$ threads.

Case 2. $\exists t' \neq t$. `t @ X[t'] = False` is *true*. Then the rule TIncCntO1 applies, and the Cubicle-$\mathcal{W}$ program reaches a state A^o. From this state, the rule

TIncCntO2 applies and performs the operation `t @ X[t'] := True`, making the program reach a state A' such that `t @ X[k] = True` is *true* for exactly $n+1$ threads.

In both cases, state A' is such that $\mathcal{T}(S', A')$ is *true*.

Inc rule with counters (from buffer)



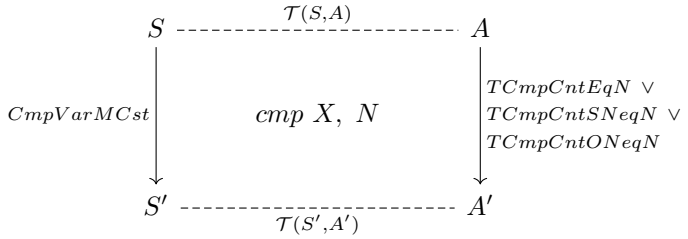
Let t be a thread and S an x86-TSO state of the form (LS, M) where $LS(t) = (eip, zf, Q, B)$ such that $B = B_1 +\!\!+ (\mathtt{X}, n) +\!\!+ B_2$ and $\mathtt{X} \notin B_1$. By rule IncVarB, the x86-TSO program can move to a state S' of the form (LS', M) where $LS'(t) = (eip', zf', Q, B')$ such that $B' = (\mathtt{X}, n+1) +\!\!+ B$ and $\forall t' \neq t. LS'(t') = LS(t')$. Let A be a state such that $\mathcal{T}(S, A)$ is *true*. In particular, the `PC[t]` register in A is equivalent to its counterpart in S, and as `X` is a counter, `t @ X[k] = True` is *true* for exactly n threads. From now on, two cases may apply :

Case 1. `t @ X[t] = False` is *true*. Then the rule TIncCntS1 applies, and the Cubicle-$\mathcal{W}$ program reaches a state A^s. From this state, the rule TIncCntS2 applies and performs the operation `t @ X[t] := True`, making the program reach a state A' such that `t @ X[k] = True` is *true* for exactly $n+1$ threads.

Case 2. $\exists t' \neq t$. `t @ X[t'] = False` is *true*. Then the rule TIncCntO1 applies, and the Cubicle-$\mathcal{W}$ program reaches a state A^o. From this state, the rule TIncCntO2 applies and performs the operation `t @ X[t'] := True`, making the program reach a state A' such that `t @ X[k] = True` is *true* for exactly $n+1$ threads.

In both cases, state A' is such that $\mathcal{T}(S', A')$ is *true*.

Cmp rule with counters (from memory)



Let t be a thread and S an x86-TSO state of the form (LS, M) where $LS(t) = (eip, zf, Q, B)$ such that $\mathtt{X} \notin B$ and $M(\mathtt{X}) = n$. By rule CmpVarMCst, the x86-TSO program can move to a state S' of the form (LS', M) where $LS'(t) = (eip', zf', Q, B)$ such that $zf' = iszero(n - N)$ and $\forall t' \neq t. LS'(t') = LS(t')$. Let A be a state such that $\mathcal{T}(S, A)$ is *true*. In particular, the `PC[t]` register in A is equivalent to its counterpart in S, and as `X` is a counter, `t @ X[k] = True` is *true* for exactly n threads. From now on, three cases may apply :

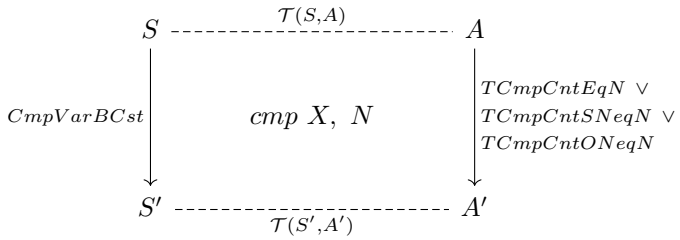
Case 1. $\forall t'$. `t @ X[t'] = True` is *true*, *i.e* $n = N$. Then the rule TCmpCntEqN applies, and the Cubicle-$\mathcal{W}$ program reaches a state A' such that `ZF[t] = 0`.

Case 2. `t @ X[t] = False` is *true*, *i.e* $n \neq N$. Then the rule TCmpCntSNeqN applies, and the Cubicle-$\mathcal{W}$ program reaches a state A' such that `ZF[t] = 1`.

Case 3. $\exists t' \neq t$. `t @ X[t'] = False` is *true*, *i.e* $n \neq N$. Then the rule TCmpCntONeqN applies, and the Cubicle-$\mathcal{W}$ program reaches a state A' such that `ZF[t] = 1`.

In both cases, state A' is such that $\mathcal{T}(S', A')$ is *true*.

Cmp rule with counters (from buffer)



Let t be a thread and S an x86-TSO state of the form (LS, M) where $LS(t) = (eip, zf, Q, B)$ such that $B = B_1 +\!\!+ (\mathtt{X}, n) +\!\!+ B_2$ and $\mathtt{X} \notin B_1$. By rule CmpVarBCst, the x86-TSO program can move to a state S' of the form (LS', M) where $LS'(t) = (eip', zf', Q, B)$ such that $zf' = iszero(n - N)$ and $\forall t' \neq t. LS'(t') = LS(t')$. Let A be a state such that $\mathcal{T}(S, A)$ is *true*. In particular, the `PC[t]` register in A is equivalent to its counterpart in S, and as `X` is a counter, `t @ X[k] = True` is *true* for exactly n threads. From now on, three cases may apply :

Case 1. $\forall t'$. `t @ X[t'] = True` is *true*, *i.e* $n = N$. Then the rule TCmpCntEqN applies, and the Cubicle-$\mathcal{W}$ program reaches a state A' such that `ZF[t] = 0`.

Case 2. `t @ X[t] = False` is *true*, *i.e* $n \neq N$. Then the rule TCmpCntSNeqN applies, and the Cubicle-$\mathcal{W}$ program reaches a state A' such that `ZF[t] = 1`.

Case 3. $\exists t' \neq t$. `t @ X[t'] = False` is *true*, *i.e* $n \neq N$. Then the rule TCmpCntONeqN applies, and the Cubicle-$\mathcal{W}$ program reaches a state A' such that `ZF[t] = 1`.

In both cases, state A' is such that $\mathcal{T}(S', A')$ is *true*.