Lazy Sequentialization for TSO and PSO via Shared Memory Abstractions

Ermenegildo Tomasco*, Truc L. Nguyen*, Omar Inverso†, Bernd Fischer‡, Salvatore La Torre§ and Gennaro Parlato*
*{et1m11,tnl2g10,gennaro}@ecs.soton.ac.uk, Electronics and Computer Science, University of Southampton, UK
†omar.inverso@gssi.infn.it, Gran Sasso Science Institute, L'Aquila, Italy
‡bfischer@cs.sun.ac.za, Division of Computer Science, Stellenbosch University, South Africa
§slatorre@unisa.it, Dipartimento di Informatica, Università degli Studi di Salerno, Italy

***Abstract*—Lazy sequentialization is one of the most effective approaches for the bounded verification of concurrent programs. Existing tools assume sequential consistency (SC), thus the feasibility of lazy sequentializations for weak memory models (WMMs) remains untested. Here, we describe the first lazy sequentialization approach for the total store order (TSO) and partial store order (PSO) memory models. We replace all shared memory accesses with operations on a shared memory abstraction (SMA), an abstract data type that encapsulates the semantics of the underlying WMM and implements it under the simpler SC model. We give efficient SMA implementations for TSO and PSO that are based on temporal circular doubly-linked lists, a new data structure that allows an efficient simulation of the store buffers. We show experimentally, both on the SV-COMP concurrency benchmarks and a real world instance, that this approach works well in combination with lazy sequentialization on top of bounded model checking.**

I. Introduction

Testing remains the most widely used approach to find program errors. It is useful when a high percentage of the selected executions lead to a violation of the program specification [26]. However, testing-only approaches such as stress testing remain highly ineffective for concurrency errors that manifest themselves rarely and are difficult to reproduce and repair [26]. Such "Heisenbugs" have become more prevalent on modern hardware architectures that use weak memory models (WMMs), because WMMs introduce additional non-determinism that remains outside the control of the testing environment. Consequently, testing needs to be complemented by automated verification techniques that can handle concurrency (and the non-determinism it introduces) symbolically.

One of these techniques is SAT/SMT-based *bounded model checking* (BMC), which has been used successfully to discover subtle errors in sequential software, even at large scale [16], [23]. Sequential BMC tools can be extended symbolically to the concurrent case by conjoining the formula representing the effect of each individual thread in isolation with a second formula representing the possible interferences caused by concurrent accesses to the shared memory [25], [4]. Since this second formula effectively includes an axiomatization of the underlying memory model, this approach can in principle work for both sequential consistency (SC) and different WMMs. However, embodying a memory model at the formula level requires extensive (and non-reusable) modifications of the underlying sequential BMC tool, and can affect scalability since the resulting expressions are large and complex.

An alternative approach is *sequentialization*, which translates concurrent programs into sequential programs with data non-determinism that (under certain assumptions) behave equivalently, so that the different interleavings do not need to be treated explicitly during the analysis. This allows the reuse of existing sequential BMC tools. Eager sequentializations [18], [27] guess the different values of the shared memory before the verification and then simulate (under this guess) each thread in turn. This means that they can explore infeasible computations that need to be pruned away afterwards. Lazy sequentializations [17] instead guess the context switch points and compute the memory. They thus only explore feasible computations and can be used as basis of very effective verification tools, such as Lazy-CSeq [14]. This is witnessed by Lazy-CSeq's top rankings in recent software verification competitions but also borne out in practice: for example, using Lazy-CSeq we discovered in 30 minutes a bug in the safestack benchmark [29], while all other approaches, including testing, failed [26]. However, to the best of our knowledge, lazy sequentializations have been developed only for SC, and not for any of the WMMs that are prevalent in modern computer architectures.

In this paper, as a first contribution, we therefore develop the first lazy sequentialization for multi-threaded programs for the total store order (TSO) [24] and partial store order (PSO) memory models. More specifically, we replace all accesses to shared memory items (i.e., reads from and writes to shared memory locations, and synchronization primitives like lock and unlock) by explicit calls to *API operations* over a *shared memory abstraction* (SMA, see Section II). For example, if `x` and `y` are two shared scalar variables then the statement $\texttt{x} = \texttt{y} + \texttt{x} + 3$ is translated into $\texttt{write}(\texttt{x}, \texttt{read}(\texttt{y}) + \texttt{read}(\texttt{x}) + 3)$. The SMA can be seen as an abstract data type that encapsulates the semantics of the underlying WMM and implements it under the simpler SC model. This isolates the WMM from the remaining concurrency aspects, and allows us to reuse existing (lazy) sequentialization techniques and tools for SC. Our approach bears some similarity to the axiomatic representation

Partially supported by EPSRC grant no. EP/M008991/1.

of memory models [25], [4] but the fundamental difference is that we work at the code level—in effect, we apply the very idea of sequentialization to WMMs themselves.

The TSO and PSO implementations of the SMA we describe here are the second contribution of the paper. They are carefully designed to optimize some parameters that lead, in combination with a lazy sequentialization targeting BMC tools, to efficient SAT/SMT encodings. Sections III and IV describe an efficient implementation of memory ADT for TSO, while Section V extends this to PSO. Section VI describes a first experimental evaluation of our approach. In particular, we compare our prototype implementation to CBMC (which implements TSO at the formula level) [4] and Niddhug [1] (which combines stateless model checking and dynamic partial order reduction). The experiments show that our approach delivers a comparable performance on simple benchmarks, but outperforms both CBMC and Nidhugg on the more complex safestack problem. It also shows that the number of timestamps (i.e., writes to the store buffers) required to expose TSO and PSO bugs is generally small. This implementation constitutes the third contribution of our paper.

II. Shared Memory Abstractions

We consider multi-threaded programs with C-like syntax including pointer arithmetics and dynamic memory allocation. We further consider POSIX threads with dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization. Thread communication is implemented via shared memory in the form of global variables. Appendix A contains details on the syntax and semantics of this class of programs.

We assume that all shared memory and synchronization operations are performed through an abstract data type that we call *shared memory abstraction* (SMA). This form can be achieved through a source-to-source transformation, if necessary. Introducing the SMA provides a separation of concerns between the shared memory and the control-flow related aspects of concurrent programs, such that the verifier design can focus on each of these aspects in isolation.

SMA API: Let us assume that shared scalar variables and threads are identified by unsigned integers. The full set of SMA API routines is:

- `init()` initializes the internal data structures of the SMA and the shared variables.
- `read(v,t)` and `read_addr(a,t)` return the value of the read issued by thread `t` of the shared variable `v` and the address `a`, respectively.
- `write(v,val,t)` and `write_addr(a,val,t)` capture the write by thread `t` of the value `val` into the shared variable `v` and the address `a`, respectively.
- `addr(v)` returns the address of the shared variable `v`.
- `malloc(expr)` dynamically allocates a number of shared memory locations given by the value of the expression `expr` and returns the base address.
- `lock(m,t)` and `unlock(m,t)` are the standard thread synchronization operations to acquire and release, respectively, a mutex `m` for thread `t`; if `m` is already acquired, the `lock` operation is blocking for `t`, i.e., `t` is suspended until `m` is released and then acquired.
- `fence(t)` flushes the store buffer of thread `t` and updates the shared memory accordingly.

SMA implementations: The semantics of concurrent programs, and in particular the concurrency aspects, can vary with the underlying memory model. For a program P, we can capture such a semantics by plugging a corresponding SMA implementation into P and thus interpreting the resulting program according to the standard interleaving semantics that assumes atomicity and sequential consistency of the memory operations (see Appendix A for more details). An SMA implementation consists of variables and data structures to capture the shared memory and functions that manipulate them, as listed in the API. Thus, we can model it as a transition system in the usual way.

III. Designing a TSO Shared Memory Abstraction

Here, we recall the TSO memory model and introduce two SMA implementations that capture it. We start with a reference implementation called TSO-SMA that represents the standard TSO semantics [24] directly but leads to complex formulas when used in a BMC-based sequentialization tool chain. We therefore introduce a new representation where the *per thread* store buffers are replaced by *per variable* write lists. We show how this, together with an indexing scheme, allows us to perform the shared memory updates implicitly, and in fact even to entirely remove any explicit representation of the shared memory. This reduces the size of the formulas for two reasons. First, BMC tools perform *function inlining* (i.e., replace each function call with the actual function code), so removing the updates, which can happen at any time and thus need to be inlined at every visible operation [17], reduces the size of the program and thus the formula. Second, because we remove the memory we do not need (propositional) variables to represent each memory write; we can instead reuse those used to represent the variable write lists. We describe an efficient shared memory abstraction eTSO-SMA that is based on this representation and is equivalent to TSO-SMA. In this section we give both SMA implementations at an abstract level; in the next section, we give the details of eTSO-SMA including concrete data structures and code for the API operations, and argue the equivalence of eTSO-SMA and TSO-SMA.

Total Store Ordering: TSO is a *relaxed-consistency* shared memory model where, different from SC, the ordering of write and read operations performed by different threads on the same variable can be altered. The behaviour of the TSO memory model can be described with respect to the architecture shown in Figure 1 [19].

Each thread t is equipped with a *store buffer* where the write operations performed by t are temporarily cached according to a FIFO policy. The effects of a cached write are visible only to the thread that has performed it. A read by a thread t of a variable y (resp. location at the address p) retrieves the value from the shared memory unless there is a cached

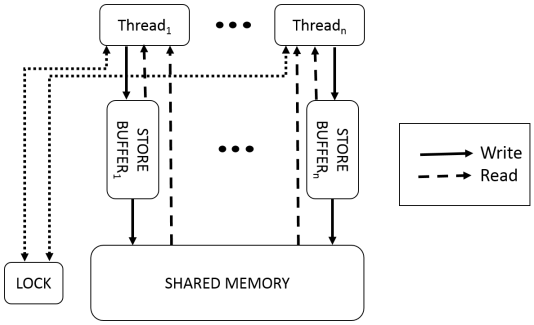

Fig. 1. TSO architecture.

write to y (resp. to the location at p) that is pending in its store buffer; in that case, the value of the most recent write in t's store buffer is returned. When a write is moved out of the store buffer the shared memory is updated accordingly. Memory *updates* can occur nondeterministically at any time in the computation provided that there are writes cached in some store buffer. Memory *fences* simply flush the store buffer of the thread executing them. A lock can be acquired only if the store buffer of the acquiring thread is empty, i.e., does not contain pending writes.

TSO-SMA: The reference implementation TSO-SMA directly simulates the behavior of the TSO memory model according to the architecture shown in Figure 1. We use an array-based queue of bounded size for each thread store buffer, and a copy of the shared variables of the initial program to store one configuration of the shared memory.

Here, the simulation of each write operation results in a small formula: we just need to encode a single element write into the queue. However, read operations lead to larger and more complex formulas. The main source of complexity is in the number of steps required to determine the most recently performed write operation still cached in the queue. All these steps need to be encoded in the formula. Similarly, a flush operation involves every element of a store buffer which again need to be encoded in the formula. The formula for simulating shared memory updates is even bigger. In fact, even though each single update requires only a constant number of steps (to dequeue the write and then modify the content of a memory location), memory updates can occur nondeterministically at each step and in the limit the writes from all store buffers can be passed into the shared memory. Therefore, memory updates require a formula of size proportional to the sum of the maximum number of elements that can be stored in each (bounded) queue. Since these updates need to be simulated at each shared memory access, this leads to a considerable blow-up of the formula size for the whole sequentialized program, rendering this direct implementation hopelessly inefficient.

Timestamping writes: The handling of memory updates can be improved by performing the dequeueing operations implicitly. For this, we keep track of the (discrete) time in the executions with a global variable `clock` and add a *timestamp* to each write that is enqueued in a store buffer. These timestamps represent the future time at which a cached

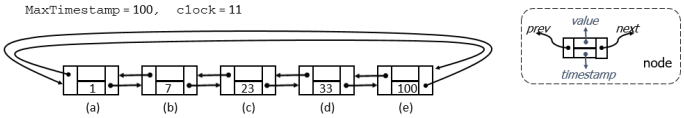

Fig. 2. T-CDLL example.

write will be used to update the shared memory. Since timestamps refer to future events and writes from different buffers can occur in any order (because the memory updates are nondeterministic), timestamps must be guessed. To ensure consistency with the TSO semantics, we must enforce that the timestamps of successive writes in the same store buffer follow a non-decreasing order. Further, a timestamp must be assigned a value that is at least the current `clock` value.

By adopting this time-stamping schema we can keep the content of a store buffer up-to-date without actually dequeuing the writes when a memory update occurs. In fact, we can just increase the value of `clock`, and all writes that have an *expired* timestamp (i.e., a timestamp that is less or equal to the value of `clock`) can be treated as removed from their respective store buffers.

eTSO-SMA: We now describe an efficient implementation where each SMA operation can be encoded with a formula of constant size and where there is no need to explicitly encode memory updates. For ease of presentation, let us assume that the original program uses only shared scalar variables and that memory locations are never accessed using their addresses. The two main ingredients of this implementation are: (1) the combined use of timestamps and the variable `clock` as above and (2) a re-arrangement of the writes of the store buffers into lists per shared variables. We refer to each such list as a *variable write list* (vw-list, for short). For each shared variable x, we denote with Q_x its associated vw-list. vw-lists contain writes as pairs (val, ts) where val is the written value and ts is the associated timestamp. A write is *expired* if its timestamp is less than or equal to the current value of `clock`.

In addition to the writes of x that are currently cached in store buffers, Q_x also contains the last write of x that has been used in a shared memory update. This gives the current value of x in the shared memory, and is the only expired write in Q_x. Thus, as sketched above, we do not need additional variables to track the shared memory. Further, we keep Q_x ordered by non-decreasing timestamps and in case of writes with the same timestamp, in the order of insertion in the list. Hence, the current valuation in the shared memory of each variable is easy to retrieve from the front of the list.

Temporal Circular Doubly Linked List: To implement the vw-lists efficiently, we introduce *temporal circular doubly linked lists* (T-CDLL, for short), which are circular doubly linked lists where:

1) nodes are of the form shown in Figure 2; fields $prev$ and $next$ contain respectively the link to the predecessor and the successor in the list as usual, and the fields $value$ and $timestamp$ contain the write;
2) there is a unique *sentinel node*; it does not correspond

to an actual write and its timestamp is maximal;
3) the sequence of nodes from the successor of the sentinel node through the sentinel node, via the $next$ link, is ordered by non-decreasing timestamps;
4) the *head* of the list is defined as the only node that (i) contains an expired write and (ii) whose successor contains a non-expired write; it is uniquely determined by `clock`.

An example of T-CDLL is given in Figure 2. There the sentinel node is (e), the head node is (b) for `clock` value 11 and would change to (c) as soon as `clock` reaches 23.

Note that the portion of T-CDLL from the head through the sentinel node ensures the properties stated for the vw-lists, and thus we can easily use this portion to store each such list. Moreover, a T-CDLL also manages the list of available nodes in the remaining part: when the value of `clock` is increased so that the current head node expires, this becomes available and its successor becomes the new head. However, the node remains linked in the list; hence, all nodes between the sentinel and the head constitute a "free list" for further writes. Caching a write only requires checking that the successor of the sentinel is expired (but not the head of the list), and if so, overwriting the values and re-linking the node.

For example, in the T-CDLL of Figure 2, only node (a) is available. By updating `clock` to 30, the head becomes (c) and (b) becomes available but remains linked to the list of available nodes that starts from (a). If we then try to cache a write with timestamp 40 in the vw-list, we can take (a), unlink it from the T-CDLL and then link it back between (d) and (e) with value and timestamp updated according to the write.

The T-CDLL for a vw-list is initialized with a fixed number of nodes that stay unchanged along the computation (i.e., the size of lists we can encode is bounded). The initial value for the timestamps is 0 except for the sentinel node, whose timestamp is set to the maximum allowed timestamp. In the next section we show how to implement T-CDLLs efficiently by using four parallel arrays, one for each node field.

IV. Implementation of eTSO-SMA

The code of eTSO-SMA comprises a *module* whose *abstract view* is essentially the API given in Section II and whose *implementation view* (i.e., the complete data type declaration along with the actual code implementing the API operations) will be discussed in this section.

Memory bounds: We assume that during the program execution all threads can access the memory, which consists of a finite sequence of locations of the same size. Each location has its own memory address which corresponds to its position in the memory. Shared variables are allocated to distinct memory locations. However, eTSO-SMA does not capture the entire memory explicitly but only tracks a bounded number of memory locations.

We use several parameters to bound our analysis and in particular the memory representation. `T` denotes the maximum thread identifier used in the program, `N` the number of nodes in each T-CDLL, `V` the maximum number of memory locations tracked along any execution (including the locations of the shared variables), and `K` the maximum timestamp.

We assume that each shared variable has an integer identifier in the range $[0, \mathtt{V}-1]$. Further, we can have an additional number of memory locations to track that can be accessed only through their memory addresses.

Auxiliary Data Structures: We use the following global auxiliary variables of type integer:

- `clock` keeps track of the number of writes performed by all threads; it is initialized to 0 and bounded by `K`.
- `address[v]` contains the physical memory address of `v`, for any `v`$\in [0, \mathtt{V}-1]$; the `init` function initializes it with distinct nondeterministic values; the values do not change during the program simulation;
- `value[v][node]` and `tstamp[v][node]` store the value and timestamp of each write associated with each location.
- `max_tstamp[t]` stores for each thread the largest timestamp of any executed write;
- `prev[v][node]` and `next[v][node]` store the link to the previous and next node in the T-CDLL for each location.
- `last_value[v][t]` and `last_tstamp[v][t]` store the last value written and the timestamp of the last write performed by each thread for each location.

The nodes of the T-CDLL corresponding to variable `v` are kept in `prev[v][i]`, `value[v][i]`, `tstamp[v][i]` and `next[v][i]` for `i`$\in [0, \mathtt{N}-1]$.

Malloc and init: During its execution a thread can require a block of n consecutive locations by invoking `malloc`(n), which returns the address of the first location of a newly allocated heap block. Memory addresses can be used to access this shared memory. Let p be a shared pointer variable and `x` be a local variable. A location with address i is *pointed to* by p if the value of p is i. Then, `*p = x` copies the value of `x` into the location pointed to by p, and statement `x = *p` copies the value of the location pointed to by p into `x`. Note that we do not represent the heap memory explicitly as we only track some of its locations. Concerning to `malloc` we maintain a bounded sequence, say of fixed size m, where each element represents a memory block. In particular, for each block we store its base address, its size, and whether it has been allocated. This sequence is implemented using arrays. The `init` function initializes each of these blocks with a nondeterministic base address and a nondeterministic size, making sure that block do not overlap. These values do not change during program executions.

Clock update: `clock_update` is a service routine that updates the variable `clock` with a nondeterministic value picked in the range from its current value to its allowed maximum value `K` (see Figure 3, lines 11-15). As a consequence of such an update, some of the writes in the T-CDLLs may expire, thus modifying some of the head nodes and therefore, the valuation of variables in the shared memory and the configuration of the T-CDLLs.

We stress that this is a very convenient way to implement the shared memory updates since we achieve this without altering the underlying data structures. Moreover, it is correct w.r.t. the TSO semantics since the writes flow into the shared memory by increasing values of the timestamps, and by the ordering we ensure on the timestamps, this enforces a correct simulation of the TSO semantics on the memory updates.

Write operation: We recall that function `write` takes as input a variable identifier `v`, a value `val`, and a thread identifier `t`. `write` first updates the clock value by invoking `clock_update` and then simulates the write operation by changing the state of the memory representation by adding a new write in the T-CDLL associated with `v`. The code for this function is given in Figure 3 at lines 17-45.

Take an available node from the T-CDLL of the variable identifier (lines 20-23). Variable `node` is set to a position of the array encoding the T-CDLL for `v` that corresponds to the successor of its sentinel node. From the property of part 3 of the definition of T-CDLLs (that we maintain as an invariant in our implementation), this is the node with the smallest timestamp in the list. We are going to use this node to cache the write. The `assume` statement at line 21 ensures that the successor of `node` is also expired. Otherwise, `node` would be the head node and thus there are no available nodes, therefore the computation must be blocked. We then remove the node from the list by appropriately setting the fields `next` and `prev` of the successor and the predecessor of `node`, respectively (lines 22-23).

Select position for insertion (lines 25-27). We nondeterministically guess a node `succ` that is the candidate to be the successor of `node` in the list (after insertion). We make sure that `succ` encodes a non-expired write by checking that its timestamp is greater than the current value of `clock`. Note that, a node with this feature always exists in since the timestamp of the sentinel node is `K`. We then set the local variable `pred` to the predecessor of `succ`. Node `pred` will be the predecessor of `node` after its reinsertion in the list.

Guess a suitable timestamp for the new write (lines 29-33). First, we guess a value (line 29), then we check that it is not smaller than the current `clock` value (line 30), and the last timestamp used for the same thread (line 31). This last check guarantees that the writes from the same thread update the shared memory in the FIFO order. The last two constraints at lines 32-33 guarantee the T-CDLL invariant that nodes must be in a non-decreasing order (part 3 of T-CDLL definition). Note that we allow the timestamp of the new write to be the same as that of the previous write in the list (which corresponds to a situation where they are passed to the shared memory in the same update). However, the inequality is strict w.r.t. the next write in the list (line 33). In this way we guarantee that a write cannot overtake another write from the same thread that is already cached in the store buffer with the same timestamp (which would violate the TSO semantics).

Node insertion at selected position (lines 35-40). We can now insert the new write into the T-DCLL. We first set its value, then its selected timestamp, and finally we insert `node`

```
 1: static uint clock;
 2: static uint address[V];
 3: static  int value[V][N];
 4: static uint tstamp[V][N+1];
 5: static uint prev[V][N+1];
 6: static uint next[V][N+1];
 7: static uint last_value[V][T];
 8: static uint last_tstamp[V][T];
 9: static uint max_last_tstamp[T];
10:
11: void clock_update( ){
12:    int tmp;
13:    assume( tmp <= K && tmp >= clock );
14:    clock = tmp;
15: }
16:
17: void write(uint v,int val,uint t){
18:   clock_update();
19:      // remove expired node from list
20:   uint node = next[v][N];
21:   assume(tstamp[v][next[v][node]]<= clock);
22:   next[v][N] = next[v][node];
23:   prev[v][next[v][N]] = N;
24:      // select position in the list for insertion
25:   uint succ = nondet();
26:   assume( succ<=N && tstamp[v][succ]>clock );
27:   uint pred = prev[v][succ];
28:      // guess suitable timestamp
29:   uint ts=nondet();
30:   assume( ts >= clock
31:           && ts >= max_last_tstamp[t]
32:           && ts >= tstamp[v][pred]
33:           && ts <  tstamp[v][succ] );
34:      // insert node at selected position
35:   value[v][node] = val;
36:   tstamp[v][t] = ts;
37:   next[v][node] = succ;
38:   prev[v][node] = pred;
39:   next[v][pred] = node;
40:   prev[v][succ] = node;
41:      // update auxiliary data
42:   max_last_tstamp[t] = ts;
43:   last_tstamp[v][t] = ts;
44:   last_value[v][t] = val;
45: }
46:
47: void write_to_address(uint a,int val,uint t){
48:      // select identifier of the memory address
49:   int v = nondet();
50:   assume( v < V );
51:   assume( address[v] == a );
52:   write( v, val, t );
53: }
54:
55: int read( uint v, uint t ) {
56:    clock_update();
57:      // retrieve the value from thread store-buffer
58:   if ( last_tstamp[v][t] > clock )
59:      return last_value[v][t];
60:      // retrieve the value from shared memory
61:   int node = nondet();
62:   assume( node < N &&
63:           tstamp[v][node] <= clock &&
64:           tstamp[v][next[v][node]]>clock);
65:   return value[v][node];
66: }
67:
68: int read_from_address( uint a, uint t ) {
69:      // selecting the id for the memory address
70:   int v = nondet();
71:   assume( v < V );
72:   assume( address[v] == a );
73:   return( read(v,t) );
74: }
75:
76: void fence( uint t ){
77:   clock_update();
78:      // make all thread's write expired
79:   if ( clock < max_tstamp[t] )
80:      clock = max_tstamp[t];
81: }
```

Fig. 3. Code stubs for eTSO-SMA.

between `prec` and `succ` (lines 37-40). Clearly, the resulting list is still a T-DCLL.

Update auxiliary data (lines 42-44). We update the auxiliary variables to ensure that their invariants are maintained.

Write-to-address operation: The first step of function `write_to_address` is to select the identifier corresponding to address, if any, and then call `write`.

Read operations: The function `read` takes as input a variable identifier `v` and a thread identifier `t`, and returns the value of `v` retrieved from the current state of the memory system. The implementation of `read` is shown in Figure 3. It first updates the clock. Then, it checks whether the last performed write by `t` on `v` has not expired yet. This condition ensures that if there is still a pending write in `t`'s store buffer, then the value of the latest such write is returned, as per the TSO semantics. Otherwise, it returns the value of the latest expired write in the T-CDLL of `v` (lines 61-65), which is always guaranteed to exist by the invariant property of T-CDLLs, and as previously argued, it corresponds to the valuation of `v` in the shared memory. When a read is performed using a memory address (lines 68-74), we first retrieve the location identifier, say `v`, corresponding to the memory address `a` and then return the value returned by calling `read` on `v`.

Fence operation: To flush the store-buffer of a thread it is sufficient to mark all its writes as expired. Thus, function `fence` shown in Figure 3 at lines 76-81 first updates the clock to the current time and then sets again the clock to the value of the maximum ticket issued by thread `t` in case it results greater than the current clock.

Correctness: We have already observed the main properties concerning the correctness of our implementation. A formal proof of this can be given by showing that the transition system T that captures eTSO-SMA is equivalent to the transition system $T_{\texttt{TSO}}$ that captures TSO-SMA (and thus the semantics of the TSO memory model) in the sense that they can simulate each other behaviors going through equivalent configurations.

The most complicated case of the proof is for the `write` function; we sketch this one here. From the observations in the write-operations section above, we have that after the execution of `write` the corresponding T-CDLL is correctly updated. Also the current write is added with a timestamp that is not smaller than the timestamp of the last previously cached write from the same thread (line 31), and in case the two timestamps are equal and the two writes are on the same variable, line 33 guarantees that we keep the same order as in the store buffer. This implies that if we start from equivalent configurations of T and $T_{\texttt{TSO}}$, then the configurations of T and $T_{\texttt{TSO}}$ resulting after the invocation of `write` are also equivalent. More details can be found in Appendix B.

Therefore, we get:

Theorem 1. *eTSO-SMA and TSO-SMA are equivalent.*

V. Extension to the PSO memory model

We recall that the semantics of PSO is the same as for TSO except that each thread is endowed with a store buffer for each shared memory location. To handle PSO we just need to modify the implementation of eTSO-SMA with the following changes in the write function from Fig. 3. In the write of a location `v` by a thread `t` we do the following:

- the guessed timestamp `ts` must be not lower than the timestamp of the last write of `t` on `v` (according to the PSO semantics, a write by a thread t of a variable following a previous write by t can overtake it, but cannot overtake a previous write of the same variable);
  line 31 of Fig. 3 must be replaced with
  ```
  && ts >= last_tstamp[v][t]
  ```
- `ts` must be the last timestamp of `t` if it is greater than the current one; line 42 of Fig. 3 must be replaced with
  ```
  max_last_tstamp[t] =
      (ts<=max_last_tstamp[t])?
          max_last_tstamp[t]:ts;
  ```

Denoting with ePSO-SMA our prototype tool obtained from eTSO-SMA by the above changes and with PSO-SMA the reference implementation for PSO (obtained similarly to TSO-SMA for TSO), we get:

Theorem 2. *ePSO-SMA and PSO-SMA are equivalent.*

VI. Experimental Evaluation

Prototype implementation: We implemented our approach for C programs with POSIX threads in a prototype tool called LazySMA[1]. It is based on the open-source CSeq framework [13], [11] which allows the development of sequentializations following a modular approach. In this framework, tools are built as pipelines of source-to-source transformations where the result of the last transformation is fed into a sequential analysis backend. For our prototype we implemented a new transformation that replaces each memory access by the corresponding operation from the API. In order to combine this new transformation with Lazy-CSeq [14], [15] we needed to "inject" the new memory management layer into a few locations where memory and concurrency handling overlap. For example, we changed the original `lock` and `unlock` simulation procedures of Lazy-CSeq to use the barriers for memory synchronization required by the weaker memory model. We used CBMC v5.3 as sequential verification backend.

Experimental set-up: We have compared our prototype implementation against two tools with built-in support for analysis under weak memory models: CBMC [12], a mature bounded model checker tool for C/C++ programs, and Nidhugg [1], a bug-finding tool that combines stateless model checking with dynamic partial order reduction on relaxed memory executions.

We ran the experiments on a dedicated machine with a Xeon W3520 2.6GHz processor and 12GB of physical memory running 64-bit linux 3.0.6. We set a 10GB memory limit and a 600s timeout for the simple benchmarks and timeout of 14,400s for the safestack example. For each tool and

[1] `http://users.ecs.soton.ac.uk/gp4/cseq/fmcad16.zip`

TABLE I
ANALYSIS RUNTIMES UNDER TSO AND PSO

	bug?	parameters							TSO runtime (*s*)			PSO runtime (*s*)		
		unwind	qsize (N)	naddr	nmalloc	bitwidth	rounds	maxclock (K)	LazySMA	CBMC	NIDHUGG	LazySMA	CBMC	NIDHUGG
dekker	•	1	2	0	0	4	2	2	0.77	0.29	**0.04**	0.75	0.25	**0.05**
lamport	•	1	2	0	0	4	2	2	0.88	0.31	**0.05**	0.88	0.29	**0.05**
peterson	•	1	3	0	0	4	2	2	0.66	0.26	**0.04**	0.65	0.25	**0.04**
szymanski	•	1	3	0	0	4	2	3	0.81	0.34	**0.07**	0.80	0.32	**0.04**
fib_longer_unsafe	•	6	2	0	0	10	6	2	**6.47**	8.19	94.84	6.51	**1.69**	135.45
fib_longer_safe		6	2	0	0	10	6	2	**9.78**	22.5	t.o.	**8.82**	31.8	t.o.
parker	•	1	2	0	0	4	2	3	1.68	0.31	**0.05**	2.19	0.28	**0.05**
stack_unsafe	•	2	2	1	2	5	2	2	1.50	0.41	**0.05**	1.49	0.35	**0.05**
litmus_safe (avg)		5	2	0	0	10	2	20	1.26	**0.17**	2.35	1.22	**0.15**	6.65
litmus_unsafe (avg)	•	5	2	0	0	10	2	20	1.27	**0.16**	3.86	1.26	**0.12**	1.58

benchmark, we set the parameters to the minimum value needed to expose the error.

Simple benchmarks: We first evaluated our approach over a set of (relatively simple) benchmarks collected from the CBMC, Poet, and Nidhugg tools, and the SV-COMP benchmark suite. The results are summarized in Table I. The unwind parameter was used by all the three tools considered in the comparison. The sum of naddr and nmalloc gives the parameter `V` as described in Section 4. The parameter bitwidth gives the size of integers (in bits) used in the sequential analysis and the parameter rounds is the number of rounds used by Lazy-CSeq.

The first block contains results for the classical mutual exclusions algorithms (`dekker`, `lamport`, `peterson`, `szymanski`). These implementations are correct under SC but not under TSO or PSO. All tools find the errors, but because of their small size, Nidhugg outperforms both CBMC and our prototype (which incurs a constant overhead for the sequentialization process) on these programs.

The second block of the table contains variations of the fibonacci-benchmark, in which two worker threads concurrently increase two shared counters, and a main thread checks whether any the two counters can reach a defined value. A full exploration of the thread interleavings is required to identify the error (or show its absence) in this program. Techniques such as partial-order reduction do not apply, and several tools struggle to analyze it. We have included both the safe and unsafe versions. Here, Nidhugg is generally slower than both CBMC and our prototype tool, and fails to terminate on the safe version. Our prototype beats CBMC on the safe cases, but is slower on the unsafe ones.

The next two benchmarks originate from industrial code: `parker` models a semaphore-like synchronization class that breaks under TSO [1] (and thus also under PSO), and `stack` which was taken from SV-COMP [7].

The last two lines show the average values for 5803 litmus tests for WMMs; note that we ran these under TSO and under PSO. For TSO, both our prototype and CBMC successfully identified the 277 test cases containing a reachable error, while Nidhugg failed to find one of them. For PSO, Nidhugg and IMU-CSeq both find 968 unsafe instances, while CBMC claims that there are 971 unsafe instances but this includes three spurious counterexamples. The performance gap between CBMC and our tool could be reduced with a more efficient implementation, as our prototype transforms each file nearly 20 times, each time requiring parsing and unparsing.

Safestack: We have conducted further experiments on a real world benchmark, Safestack [10], which is a lock-free stack implementation designed for weak-memory models. It contains a rare bug that is hard to find with automatic bug-finding techniques already under SC (including random testing, Nidhugg, CIVL [32], CBMC and other approaches based on BMC) [26]. The only tool we are aware of that can automatically find a genuine counter-example is Lazy-CSeq. It requires a minimum of 4 threads, 3 loop unwindings, and 4 rounds of computation to expose a bug caused by two of these threads simultaneously modifying the element at the first position of the array implementing the queue. This shows that the error is actually quite deep, which explains why other approaches based on explicit handling of interleaving fail.

Safestack is written in C++. We manually translated it into C, providing simulation functions for the C11 atomic functions used in the test. We experimented with this C version, with different bounds for the queue size of each memory address, and the maximal timestamp along any bounded computation. Table II summarises these experiments. Note that we only used three bits to represent integers during the analysis. We then checked whether counterexamples found also hold for full 32-bits integers, by running Lazy-CSeq over the exact schedule extracted from the counterexample. A "Yes" entry in the CEX? column means that the counterxample holds, thus there are no spurious counterexamples (due to overflow).

Because SC and TSO coincide if maxclock is set to 1, the first four lines indirectly show the overhead paid for our TSO encoding. Since the SC analysis using Lazy-CSeq (not shown here) requires approximately 3 minutes, the TSO encoding itself thus introduces an approximately 3x-4x overhead. The last two lines show that we can still find the error under "proper" TSO. It also shows that the weaker memory model reduces to 3 (from 4) the number of rounds required to expose this error; however, the analysis time grows noticeably, by almost an order of magnitude. Finally, increasing maxclock (for fixed values of qsize and rounds) shows that the analysis explores more reorderings of reads over writes (witnessed by the increased memory consumption).

VII. RELATED WORK

The transformation of concurrent programs under TSO into equivalent programs under SC is intrinsic in the architecture from Figure 1. However, the explicit modeling of the store buffers in the resulting program introduces a substantial overhead for standard SC verification tools.

In [5], the authors replace the store buffers with $O(k)$ local variables per thread, where k is the number of context-switches for each thread that is allowed in the analysis. The main intuition there is: when a write operation occurs, a future context number is guessed with the meaning that the write will

TABLE II
ANALYSIS RUNTIMES FOR SAFESTACK UNDER TSO AND PSO

parameters			TSO analysis (3 bits)			CEX check (32 bits)		PSO analysis (3 bits)		CEX check (32 bits)	
K	N	rounds	Time	Mem.	Reach?	CEX?	Time	Time	Reach?	CEX?	Time
1	2	4	10m18s	0.8GB	Yes	Yes	23s	11m42s	Yes	Yes	4.82s
1	2	3	12m2s	0.6GB	No	-	-	11m16s	No	-	-
1	3	4	13m45s	1.2GB	Yes	Yes	30s	21m6s	Yes	Yes	6.40s
1	3	3	12m50s	0.9GB	No	-	-	12m20s	No	-	-
3	2	4	26m55s	1.4GB	Yes	Yes	24s	20m47s	Yes	Yes	4.33s
3	2	3	24m34s	1.0GB	No	-	-	27m15s	No	-	-
3	3	4	74m22s	3.4GB	Yes	Yes	31s	31m16s	Yes	Yes	5.47s
3	3	3	62m22s	1.0GB	Yes	Yes	30s	20m7s	Yes	Yes	2.84s
3	3	2	12m14s	0.6GB	No	-	-	11m14s	No	-	-
7	2	4	47m17s	2.4GB	Yes	Yes	27s	104m35s	Yes	Yes	6.05s
7	2	3	35m7s	1.3GB	No	-	-	36m14s	No	-	-

be visible to the other threads at that context. This is similar to our guess of a future timestamp in the sense that it implicitly gives a total ordering of the shared memory updates, but in our setting this is completely unrelated to the thread context at which the memory update will occur.

In [30], the authors replace the store buffers by embedding each of them symbolically in the thread locations. Their translation goes through the construction of the corresponding transitions systems that seems appropriate for a backend as SPIN but in our experiments works poorly with BMC since it introduces a lot of redundancy in the constructed formulas.

Another main difference of our approach with the above-mentioned two is that we rearrange the contents of store buffers *per variable* instead of *per thread* and entirely maintain them in T-CDLLs of bounded size. This, along with the strong invariant properties of T-CDLLs, results in smaller formula encodings computed by the BMC backend tools (see Section III).

Other recent work that have dealt with the verification of concurrent programs under weak memory model semantics are [1], [2], [3], [4], [6], [8], [9], [28], [31]. The most related to ours is [3] where the authors give a general reduction technique to SC by augmenting the programs with arrays to simulate the caching and buffering due to the weak memory models and use it in combination with CBMC. In [4], CBMC is enhanced with a reduction based on partial orders.

We have designed our translation to target BMC backends and used the tool Lazy-CSeq [14], [13] for our experiments. Lazy-CSeq implements an efficient lazy sequentialization of concurrent programs that works exceptionally well with BMC backends and has won the SV-COMP twice [7]. It performs a bounded context-switching analysis [21] and has been recently extended to unbounded programs [20]. The idea of sequentialization was originally proposed in [22] but became popular with the first scheme for an arbitrary but bounded number of context switches [18].

VIII. CONCLUSIONS

In this paper we have described a new approach to the verification of concurrent programs under weak memory models. We have introduced an abstract data type that factors out the semantics of the memory model, allowing us to reuse tools designed for the analysis of concurrent programs under SC. We have given an efficient implementation of the ADT that works well in combination with Lazy-CSeq. We have demonstrated the effectiveness of this approach for finding bugs under TSO and PSO: our prototype tool is competitive with existing tools on standard benchmarks used in the literature; it also works for more complex benchmarks that are, to the best of our knowledge, out of reach for existing bug-finding approaches. We have also observed that the designed translation puts a reasonable overhead on the backend tool. We have developed our approach for TSO memory model, but a simple extension lets us also handle the PSO memory model. The main change was to organize the cached writes per variable and thread, and not just per variable. We believe that our approach can also be extended to further weak memory models, and leave this for future work.

REFERENCES

[1] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. F. Sagonas. Stateless model checking for TSO and PSO. In *TACAS*, pages 353–367, 2015.

[2] T. Abe and T. Maeda. A general model checking framework for various memory consistency models. In *PDP*, pages 332–341, 2014.

[3] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, pages 512–532, 2013.

[4] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, pages 141–157, 2013.

[5] M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, pages 99–115, 2011.

[6] M. F. Atig, A. Bouajjani, and G. Parlato. Context-bounded analysis of TSO systems. In *FPS*, pages 21–38, 2014.

[7] D. Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *TACAS*, pages 401–416, 2015.

[8] A. Bouajjani, G. Calin, E. Derevenetc, and R. Meyer. Lazy TSO reachability. In *FASE*, pages 267–282, 2015.

[9] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PDLI*, pages 12–21, 2007.

[10] G. Chen, H. Jin, D. Zou, B. B. Zhou, Z. Liang, W. Zheng, and X. Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Trans. Dep. Sec. Comput.*, (6):368–379, 2013.

[11] B. Fischer, O. Inverso, and G. Parlato. CSeq: A Concurrency Pre-processor for Sequential C Verification Tools. In *ASE*, pages 710–713, 2013.

[12] A. Horn and D. Kroening. On partial order semantics for sat/smt-based symbolic encodings of weak memory concurrency. In *FORTE*, pages 19–34, 2015.

[13] O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In *ASE*, pages 807–812, 2015.

[14] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *CAV*, pages 585–602, 2014.

[15] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Lazy-cseq: A lazy sequentialization tool for C - (competition contribution). In *TACAS*, pages 398–401, 2014.

[16] D. Kroening and M. Tautschnig. Automating software analysis at large scale. In *MEMICS*, pages 30–39, 2014.

[17] S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, pages 477–492, 2009.

[18] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Meth. in Sys. Des.*, (1):73–97, 2009.

[19] A. Morrison and Y. Afek. Temporally bounding TSO for fence-free asymmetric synchronization. In *ASPLOS*, pages 45–58, 2015.

[20] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In *ATVA, to appear, http://eprints.soton.ac.uk/397033/*, 2016.
[21] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pages 93–107, 2005.
[22] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.
[23] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller. Successful use of incremental BMC in the automotive industry. In *FMICS*, pages 62–77, 2015.
[24] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, (7):89–97, 2010.
[25] N. Sinha and C. Wang. On interference abstractions. In *POPL*, pages 423–434, 2011.
[26] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: an empirical study. In *PPoPP*, pages 15–28, 2014.
[27] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Verifying concurrent programs by memory unwinding. In *TACAS*, pages 551–565, 2015.
[28] E. Tomasco, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Separating computation from communication: A design approach for concurrent program verification. In *Technical Report, http://eprints.soton.ac.uk/397759/*, 2016.
[29] D. Vyukov. Bug with a context switch bound 5, 2010.
[30] H. Wehrheim and O. Travkin. TSO to SC via symbolic execution. In *HVC*, pages 104–119, 2015.
[31] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *PLDI*, pages 250–259, 2015.
[32] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel. CIVL: formal verification of parallel programs. In *ASE*, pages 830–835, 2015.

$$
\begin{array}{lcl}
P & ::= & (dec;)^*\ (type\ f\ (\langle dec,\rangle^*)\ \{(dec;)^*\ stm\})^* \\
dec & ::= & type\ z \mid type*\ p \\
type & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{void} \\
stm & ::= & seq \mid conc \mid \{\langle stm;\rangle^*\} \\
seq & ::= & \texttt{assume}(b) \mid \texttt{assert}(b) \mid x = e \mid f(\langle e,\rangle^*) \mid \texttt{return}\ e \\
 & & \mid \texttt{if}(b)\ \texttt{then}\ stm\ \texttt{else}\ stm \mid \texttt{while}(b)\ \texttt{do}\ stm \\
conc & ::= & p = \texttt{addr}(y) \mid p = \texttt{malloc}(e) \mid y = x \mid *p = x \mid x = y \mid x = *p \\
 & & \mid t = \texttt{create}\ f(\langle e,\rangle^*) \mid \texttt{join}\ t \mid \texttt{fence} \mid\ \texttt{lock}\ m \mid \texttt{unlock}\ m
\end{array}
$$

Fig. 4. Syntax of multi-threaded programs.

APPENDIX A
SHARED-MEMORY MULTI-THREADED PROGRAMS

a) Syntax: We consider multi-threaded programs with C-like syntax including pointer arithmetics and dynamic memory allocation. We also consider POSIX threads with dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization. Thread communication is implemented via shared memory in the form of global variables.

The syntax of programs is defined by the grammar shown in Figure 4. Terminal symbols are set in typewriter font. $\langle n\ \texttt{t}\rangle^*$ represents a possibly empty list of non-terminals n that are separated by terminals `t`; x denotes a local variable, y a shared variable, p a pointer variable, m a mutex, t a thread variable and f a function name. We assume expressions e to be local variables, pointer variables, and integer constants that can be combined using mathematical operators. Boolean expressions b comprise the constants `true`, `false`, and Boolean variables, and can be combined using standard Boolean operations.

A *multi-threaded* program consists of a list of *global* variable declarations (i.e., *shared* variables), followed by a list of functions. Each function has a list of zero or more typed parameters, and its body has a declaration of *local* variables followed by a statement.

A statement is either a sequential or a concurrent statement, or a sequence of statements enclosed in braces (*compound statement*). A *sequential statement* can be an `assume`- or `assert`-statement, an assignment, a call to a function that takes multiple parameters (with an implicit call-by-reference parameter passing semantics), a `return`-statement, a conditional statement, or a loop. All variables involved in a sequential statement must be local. A *concurrent statement* can be a concurrent assignment (i.e., an assignment involving a location in the shared memory and local variable), or a call to a thread routine, such as a thread creation or join, or a thread synchronization primitive, such as a fence, or a mutex operation.

We assume that a valid program P satisfies the usual well-formedness and type-correctness conditions. We also assume that P contains a function `main`, which is the starting function of the only thread that exists in the beginning. We call this the *main thread*. We further assume that there are no calls to `main` in P and that no other thread can be created that uses `main` as starting function.

b) Semantics: We adopt an interleaving semantics where only one of the *enabled* threads can be *active* at any given time. Initially, only the *main thread* is enabled and thus active; new threads can be spawned from any thread by invoking `create`. Once created, a thread is added to the pool of the enabled threads. At a *context switch* the currently active thread is suspended, and one of the enabled threads is resumed and becomes the active thread. When a thread becomes active for the first time it starts from the beginning of its start function, otherwise it resumes from the point where it was suspended. For ease of presentation, we assume that each statement is atomic.

A *thread configuration* is a triple $\langle locals, pc, stack\rangle$, where *locals* is a valuation of the local variables, *pc* is the *program counter* that tracks the currently executing statement, and $stack$ is a stack of function calls that works as usual. In the SC memory model a *configuration* of a multithreaded program is a tuple of thread configurations along with valuation of the global variables that are shared by all threads. In the TSO and PSO memory models the notion of configuration is extended to include the valuations of the store buffers.

The formal semantics is given by a transition system that use configurations as states and the transitions update the configurations according to the given informal semantics along with a standard C-like semantics for the sequential statements.

Appendix B
Proof of Theorem 1

In this section, we show that the implementation eTSO-SMA based on the notion of vw-lists correctly captures the semantics of the TSO memory model. For this, we prove that the implementation schemes eTSO-SMA and TSO-SMA are equivalent in the sense that they can simulate each other step-by-step going through equivalent configurations as formalized in Section II.

We start by describing the transition systems T and $T_{\texttt{TSO}}$ that capture the semantics of eTSO-SMA and TSO-SMA respectively.

A store buffer for a thread t is a sequence of zero or more tuples of the form (v, val, t) where v and val are respectively the location and the value of the write. According to the architecture from Figure 1, a *TSO memory configuration* is a tuple of the form $(\nu, \langle b_t\rangle_{t\in Th}, Ter)$ where ν is a valuation of the shared memory, $\langle b_t\rangle_{t\in Th}$ denotes a tuple of valuations of store buffers b_t for each thread $t \in Th$, and $Ter \subseteq Th$ is the set of terminated threads.

The transition system $T_{\texttt{TSO}}$ that captures the semantics of TSO-SMA is defined as follows. The set of states $S_{\texttt{TSO}}$ is the set of the TSO memory configurations augmented with a new state that is taken as the initial state. Transitions are labeled with the corresponding action, i.e., the call of a function from the API or a thread creation/join/termination actions. From the initial state only transition that correspond to the init function can be taken, and the effect is to reach a TSO memory configuration that gives the initial valuation to the shared locations and has only one store buffer that is empty (the store buffer of the main thread, that is the only one created in the beginning). From each TSO memory configuration there is a transition that adds an empty store buffer for a thread creation and a transition that adds a thread from Th to Ter for thread termination. Transitions over a join are allowed only from configurations where the waited thread is terminated. When a thread is terminated no more actions from this thread are allowed. The other transitions are defined for the remaining API functions and according to the TSO semantics given in Section III. For example, a write of v with value 3 by thread t_1 from a state $(\nu, \langle b_t\rangle_{t\in Th})$ is captured by a transition to a state $(\nu, \langle b'_t\rangle_{t\in Th})$ where $b'_t = b_t$ for $t \neq t_1$ and $b'_{t_1} = b_{t_1}.(v, 3, t_1)$. Similarly, a read of v by t_1 that returns 5 is captured by a self-loop onto all the states of the form $(\nu, \langle b_t\rangle_{t\in Th})$ such that either the last write of v cached in b_{t_1} gives value 5 or, in case there is no such write in b_{t_1}, ν evaluates v to 5. Dynamic memory allocation extends the valuation ν with new locations.

Analogously, one can define the semantics of eTSO-SMA by a transition system T according to the description given in Section III. In particular, a valuation of a vw-list for a location v is a sequence of tuples of the form (v, val, ts, t) where val is the value assigned to v, ts is the associated timestamp and t is the thread that has performed the write (note that for the ease of presentation, differently from the description given in Section III, we annotate into each tuple also the thread and the location name). Then, the set of states S of T is the set of all the tuples of the form $(d, Th, Ter, \langle q_v\rangle_{v\in Var})$ where d is the current timestamp, Th is the set of created threads, $Ter \subseteq Th$ is the set of terminated threads and q_v is a valuation of the vw-list of location $v \in Var$. We observe that for dynamic memory allocation, we add a new vw-list for each newly created location and thread creation just adds the newly created thread to Th.

We recall that a run of a transition system is a sequence of transitions $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots s_h$ where s_0 is an initial state and for each $i = 0, \dots, h-1$, $s_i \xrightarrow{a_{i+1}} s_{i+1}$ is a transition from s_i to s_{i+1}.

Informally, two SMAs are equivalent if they can simulate each other step-by-step going through equivalent configurations. Precisely, given two SMAs, denote with T_i the respective transitions system, and let S_i be the set of states of T_i, for $i = 1, 2$. We say that T_1 is *equivalent* to T_2 if there is a function λ that maps states from S_1 to states from S_2 and for each sequence of actions $a_1 \dots a_h$:

- for each run $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots s_h$ of T_1, there is a run $s'_0 \xrightarrow{a_1} s'_1 \xrightarrow{a_2} \dots s'_h$ of T_2 such that $\lambda(s_i) = s'_i$ for $i = 1, \dots, h$;
- for each run $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots s_h$ of T_2, there is a run $s'_0 \xrightarrow{a_1} s'_1 \xrightarrow{a_2} \dots s'_h$ of T_1 such that $\lambda(s'_i) = s_i$ for $i = 1, \dots, h$.

Note that the above notion of equivalence ensures that the two SMAs update the memory consistently on every sequence of concurrent statements that can be issued in a multithreaded

program. Therefore, they can be used interchangeably without altering the semantics of the programs.

Since TSO-SMA is the SMA reference implementation for the TSO memory model, we get that any implementation that is equivalent to it correctly captures the TSO memory model.

We can now show Theorem 1.

Theorem 1. *eTSO-SMA and TSO-SMA are equivalent.*

Proof. We start by defining a function λ.

For each state $s = (d, Th, Ter, \langle q_v \rangle_{v \in Var}) \in S$, we define the corresponding TSO memory configuration $\lambda(s)$ as the state $(\nu, \langle b_t \rangle_{t \in Th}, Ter) \in S_{\texttt{TSO}}$ such that $|\nu| = |Var|$ and:

- ν is the valuation that assigns to each location v the value given by the head of the corresponding list q_v, i.e., for each shared location v, the head of the vw-list is tuple of the form $(\nu[v], ts, t)$ where $\nu[v]$ denotes the valuation of v by ν;
- each store buffer b_t is the sequence of writes by thread t taken from the tails of the lists q_v for any $v \in Var$ and ordered according to the timestamps; more precisely, let α be the minimal sequence such that each q'_v, $v \in Var$, is a subsequence of α where $q_v = (v, val_v, ts_v, t_v).q'_v$ (i.e., q'_v is the tail of q_v); for each thread $t \in Th$, b_t is the maximal subsequence of α containing writes by t (modulo projecting out t from the tuples).

We also set $\lambda(s_{in})$, where s_{in} is the initial state of T, to the initial state of $T_{\texttt{TSO}}$.

The "if" direction directly follows from the fact that the relation $\{(s, \lambda(s)) \mid s \in S\}$ is a simulation. This can be shown by case inspection and follows the description given in Section III.

For the "only if" direction, we fix a run ρ of $T_{\texttt{TSO}}$. From this run, we assign increasing timestamps to the write operations according to the order in which they occur in the run. Then, we simulate step-by-step the transitions of ρ in T by choosing at the write operations the timestamps computed above. Note that in T the timestamps are guessed nondeterministically and are bounded from below by the current timestamp, thus the computed timestamps can be selected. By case inspection and according to the description given in Section III, we can show that this way from any run $s_0 s_1 \dots s_h$ of $T_{\texttt{TSO}}$ we can compute a run $s'_0 s'_1 \dots s'_h$ of T such that $\lambda(s'_i) = s_i$ for $i = 1, \dots, h$. □