Verification of a lazy cache coherence protocol against a weak memory model

Christopher J. Banks[1], Marco Elver[1*], Ruth Hoffmann[2], Susmit Sarkar[2], Paul Jackson[1], Vijay Nagarajan[1]
[1]University of Edinburgh, [2]University of St Andrews

Abstract—In this paper, we verify a modern lazy cache coherence protocol, TSO-CC, against the memory consistency model it was designed for, TSO. We achieve this by first showing a weak simulation relation between TSO-CC (with a fixed number of processors) and a novel finite-state operational model which exhibits the laziness of TSO-CC and satisfies TSO. We then extend this by an existing parameterisation technique, allowing verification for an unbounded number of processors. The approach is executed entirely within a model checker, no external tool is required and very little in-depth knowledge of formal verification methods is required of the verifier.

I. INTRODUCTION

In parallel architectures with local caches, cached values can become stale. Therefore, it is imperative that the system guarantees shared memory correctness by ensuring that it correctly implements a *memory consistency model* (MCM)—the formal model that determines what value a read should return [1]. An integral component of enforcing an MCM is the *cache coherence protocol* (CCP), which is responsible for making writes visible to other caches in an order that is consistent with the MCM.

Traditionally, CCPs have been designed for the strictest of MCMs—*sequential consistency* (SC). Previously, this has been beneficial as a way to decouple the design of a CCP from the MCM; indeed, a CCP designed for the strongest of MCMs could bolt-on to other weaker MCMs. Unfortunately, this simplicity comes at a cost.

The strict program order requirements of SC mandates that writes are made globally visible before any subsequent memory operation from the same processor. To guarantee this, CCPs *eagerly* invalidate other non-local shared copies upon a write. In effect, such eager CCPs enforce the Single-Writer–Multiple-Reader (SWMR) invariant [2]—a cache line may only have either a single writer or multiple readers. To this end, eager CCPs must maintain a vector of processors sharing a cache line, but this vector scales linearly with the number of processors [3], [4]. Thus these protocols do not scale well to large-scale many-core processors.

Luckily, modern architectures tend to have more relaxed MCMs like Total Store Order (TSO)—used in prevalent architectures such as x86 and SPARC. Consequently, it is possible for CCP designers to take advantage of these relaxations. Indeed, there has been significant recent research on *lazy coherence protocols* [5], [3], [6], [4], that exploit the fact

that relaxed models only require memory to be consistent at synchronisation boundaries. In these protocols, shared lines are *self-invalidated* on synchronisation boundaries and therefore no longer require a (poorly scaling) sharing vector.

This poses a problem for the verification of such protocols. Traditionally, formal verification approaches for CCPs [7], [8] have focused on model checking protocol-specific safety properties such as the SWMR invariant [2]. However, these new lazy CCPs that are designed to take advantage of weak MCMs violate SWMR by design and hence cannot be verified in the usual way. They need to be verified in a stronger manner: for adherence to the MCM. This is especially appropriate for the protocol we study, TSO-CC [6], because it was designed specifically with the TSO memory model in mind.

Challenges: If these new scalable lazy CCPs are to see the light of day, we believe they need to be formally verified against the MCM. A testing approach does not cover all corner cases and does not give the confidence that formal verification brings. Equally, the subtlety of behaviours exhibited by both lazy CCPs and weak MCMs warrants a rigorous approach. Only formal verification will suffice to allay skepticism surrounding the behaviour of lazy CCPs. Furthermore, the verification technique should be generally applicable, should not assume the verifier to have sophisticated knowledge beyond the protocol, and it should scale to many-core processors.

Our result: In this paper, for the first time, we formally and exhaustively verify a modern lazy CCP against the MCM which it is supposed to implement. Our protocol of interest is TSO-CC (Section II), a scalable lazy CCP which was designed to target TSO. We establish our result for fixed cache sizes, but for any number of processors. Our verification focuses on safety; we do not tackle liveness. This enables our verification approach to use a slightly abstract version of CCP where, for example, access counters are not modelled explicitly.

Our approach to verification proceeds as follows. First, we propose a novel finite-state operational model TSO-LB, based on load buffers, that abstracts our lazy CCP TSO-CC. Second, we use a model checker to establish that TSO-CC is a refinement of the TSO-LB operational model. Initially we show refinement for a fixed number of processors; subsequently we deploy the parameterised verification technique of Chou et al. [9] to extend our refinement result to an arbitrary number of processors. Finally, we show that the TSO-LB operational model is stricter than an axiomatic specification of TSO.

Contributions: Our approach is inspired by Chatterjee et al. [10], who showed how CCPs can be verified against their

MCMs using a model checker. Beyond this work, we make a number of specific advances.

First, we support, for the first time, a lazy CCP through the use of a novel abstract operational model. A lazy CCP like TSO-CC *pulls* new values via self-invalidates upon a read, in contrast to conventionally eager CCPs which *push* invalidates upon a write. The nature and timing of invalidations in eager and lazy CCPs are different. Current operational models abstract the push-based invalidates, which makes it difficult to show that lazy CCPs refine them. We therefore needed to introduce this novel operational model we call TSO-LB which abstracts pull-based self-invalidates.

Second, we provide a proof that our TSO-LB model satisfies an axiomatic characterisation of TSO, however in Chatterjee et al. the task of showing the abstract operational models are consistent with axiomatic descriptions of the MCMs is not completed (and, as far as we can tell, was never subsequently completed). In our case, the proof is particularly important given how TSO-LB differs from conventional operational models for TSO.

Third, we employ the parameterisation technique of Chou et al. [9] to verify for an arbitrary number of processors (whereas Chatterjee et al. only verified for a fixed number of processors). In doing so, we demonstrate that the technique is not only useful when model checking CCP properties, but also is useful when using model checking to verify refinement and show a CCP satisfies the relevant MCM.

Other related work: Another alternative approach by Manerkar et al. [11] uses CCICheck, which explores ordering relations between CCP and MCM; however, protocols must be described in an axiomatic style—orthogonal to typical operational descriptions of protocols—and verification is with respect to specific litmus tests—which may not capture every MCM behaviour and hence not exhaustive. It is notable that these approaches only verify for a fixed number of processors; an approach to solving this problem is found in compositional model checking approaches pioneered by McMillan [12]. This method was further refined by Chou et al. [13], [9] and made practical; however, they, once again, only deal with protocol-specific properties. Likewise, Pong and Dubois [14], [15] verify compositionally, using Symbolic State Models, but again only against protocol-specific properties.

Abdulla et al. [16] recently propose the Dual-TSO operational model for TSO for program verification, in which they replace the store buffer in the traditional operational model with a load buffer. However, their notion of a load buffer has unbounded queues with potentially multiple values for an address, and thus does not help us with the infinite state-space problem. Our model also works very differently (but similar to CCP's like TSO-CC) by propagating multiple addresses to a load buffer atomically. So our model is not obviously a refinement of some finite restriction of the Dual-TSO model. It is also worth noting that we first defined our TSO-LB model [17] concurrently with Abdulla et al.

II. TSO-CC

TSO-CC [6] is a lazy CCP, designed to address the scalability issues surrounding CCPs for large numbers of cores.

Lazy CCPs, like TSO-CC, take account of the fact that the relaxed memory models employed in modern multi-core processors only require memory to be consistent at synchronisation boundaries. Consequently, instead of eagerly enforcing coherence at every write, coherence is enforced lazily only at synchronisation boundaries. Thus, upon a write, data is merely written to a processor-local write-buffer, the contents of which are flushed to the shared cache upon a *release*. Upon an *acquire*, shared lines in the local caches are self-invalidated—thereby ensuring that reads to shared lines fetch the up-to-date data from the shared cache. In effect, the CCP may be much simpler and *does not require a sharing vector*.

However, the design of TSO-CC is specifically directed by the TSO memory model which has no explicit release or acquire instructions. It follows that, as reads have acquire semantics and writes have release semantics, a TSO compliant CCP would only need to consider each read/write an acquire/release; this, of course is not efficient because all reads and writes would need to be propagated, effectively negating the provision of local caches.

The approach in TSO-CC is that for each cache line in the shared cache, it keeps track of whether the line is exclusive, shared, or read-only. Shared lines do *not require tracking of sharers* (making TSO-CC more scalable than standard directory-based protocols). Additionally, for exclusive cache lines, it only maintains a pointer to the owner.

Since it does not track sharers, writes do not eagerly invalidate shared copies in other processors. On the contrary, writes are merely propagated to the shared cache in program order (thus ensuring write-write order). To save bandwidth, instead of writing the full data block to the shared cache, it merely propagates the coherence states. Intuitively, the *most recent* value of any data is maintained in the shared cache.

Reads to shared cache lines are allowed to read from the local cache, up to a predefined number of accesses (potentially causing a stale value to be read), but are forced to re-request the cache line from the shared cache after exceeding an access threshold (the implementation maintains an access counter per line). This ensures that any write (used as a release) will eventually be made visible to the matching acquire, ensuring *eventual write propagation*. When a read misses in the local cache, it is forced to obtain the most recent value from the shared cache. In order to ensure the read-read order, future reads will also need to read the most recent values. To guarantee this, whenever a read misses in the local cache, it self-invalidates all shared cache lines. Finer details of the protocol may be found in the original paper by Elver and Nagarajan [6]. It should be noted that our model implements the basic protocol, without timestamps.

Prior TSO-CC verification work: In order to check that the protocol implementation adheres to TSO, the original authors of TSO-CC used the diy [18] tool to generate litmus tests for TSO (according to the method detailed in Owens et al. [19]) and ran it in a full-system simulator. An independent approach to verification was made by CCICheck [11], using TSO-CC as a case study. CCICheck uses abstract axiomatic models of pipeline and memory system, and verifies that a set of litmus tests is not violated. However, whilst a litmus test

based approach provides some confidence that the protocol is correct, it is by no means an exhaustive means of verification and corner cases may be missed. In order to minimise the potential for missed corner cases in a detailed cycle-accurate full-system implementation, Elver and Nagarajan developed McVerSi [20], a test generation framework for fast memory consistency verification in simulation. This approach, whilst it further increased confidence and testing of corner cases, is still not exhaustive. The remainder of this paper solves this problem with an entirely exhaustive approach to verifying TSO-CC against the TSO memory model.

III. TSO-CC SATISFIES TSO

In this section, we show that the lazy cache coherence protocol TSO-CC does indeed satisfy the constraints of the TSO memory consistency model. This solves the problems associated with the previous verification approaches: corner cases which could be missed by insufficient testing would now be revealed by exhaustive exploration of the state space. For now, we only show that this is true for the simpler case of a fixed number of processors. We go on to show, in Section IV, that this is true for a parameterised model of TSO-CC with any number of processors.

We took a number of discrete steps in the process of verifying the protocol. The first step was to translate the protocol into a suitable model for verification. For this purpose we chose the Murφ language and model checker [21]. Murφ is a well-established model checker and extensively used in both previous academic studies [21], [10], [22] and in industry [12], [7], [9], [23], [3]. We then went on to show that this model satisfied some basic properties, such as freedom from deadlock, using the model checker. Our approach to this is detailed in Section III-A.

The next step in the process was to show that the TSO-CC model satisfied the constraints of TSO. One way to achieve this was to show there exists a *weak simulation relation* between TSO-CC and an operational model of TSO. A weak simulation relation exists if the observable actions (reads/writes to a memory location) in the CCP model can be matched by actions in the model of TSO. This concept is defined more formally in Section III-D, in which we also explain our approach to showing weak simulation using the model checker.

However, in order for our approach to work, we needed an operational model of TSO. Such models exist in the literature but tend to be *store buffer based* [19], [24]. These models, while abstracting push-based eager CCPs well, make it difficult to show that lazy CCPs (which pull new values via self-invalidates) refine them. Furthermore, whereas such models require unbounded store buffers, we needed a finitely enumerable model for use with a model checking approach. Hence, in Section III-B we define TSO-LB, a *load buffer based* operational model with bounded buffers that abstracts lazy CCPs. After establishing that TSO-LB exhibits only TSO behaviour, we were able to use the operational model as part of our verification strategy.
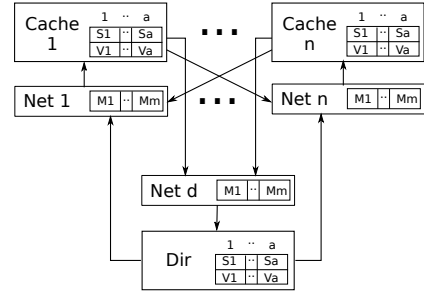

Fig. 1. Concrete model structure.

A. Model checking in Murφ

We began by defining a Murφ model of the TSO-CC protocol. The model implements the basic TSO-CC protocol as described in the original paper [6], with each rule in the protocol description relating to a rule in the Murφ model; it has parameters for the number of processors, number of addresses, and number of values; the model was checked using three address locations and two values. The model is a faithful implementation of the protocol with the only abstraction being the abstract interpretation of the access counter—as described below. The model is constructed as a set of caches and a directory, each having a state and a set of addresses or memory locations, each with a set of possible values. The interconnection network is represented as a set of sets of messages; each node (cache or directory) can write or read to or from the network (Figure 1).

A set of rules, each a *guard $\implies$ action* pair, then defines the behaviour of the model. As an example, the following is a pair of sample rules taken from the full ruleset[1]:

$$c[a].\mathtt{state} = \mathtt{I} \implies \mathtt{SendGetS}(c, Dir, a);$$
$$c[a].\mathtt{state} := WS \qquad \text{(Read I)}$$

$$c[a].\mathtt{state} = \mathtt{E} \implies c[a].\mathtt{val} := v$$
$$c[a].\mathtt{state} := M; \qquad \text{(Write E)}$$

where c is a cache, a is an address (memory location), v is a value, and $c[a].\mathtt{state}$ (or $c[a].\mathtt{val}$) is the state (or value) for the given cache and address. The first rule (Read I) is the Read rule for the Invalid cache state and the second is the Write rule for the Exclusive cache state. When a cache is in state I and does a Read, it sends a GetS message to the directory and switches to state WS. When a cache is in state E it may do a Write, store the written value, and switch to state M (Modified). The function SendGetS handles the passing of a GetS message to the network.

We then define a rule that handles the receipt of messages from the network at each node (cache or directory). Within this rule are some functions which handle actions performed when a message is received; the following is an extract from the DirectoryReceive function for handling the messages in the previous example:

---

[1]The full ruleset can be found at https://github.com/icsa-caps/tso-cc

```
DirectoryReceive(msg, a) =
        if Dir[a].state = I ∧ msg.type = GetS
        then SendDataS(msg.src, a, . . .);
             ReplaceOwner(msg.src, a);
             Dir[a].state := WE1
        else . . .
```

There is a function `CacheReceive` which has similar conditions for receiving messages at a cache.

Another pair of rules which are of interest are the Read rules for the Shared cache state. Part of the lazy invalidation scheme for TSO-CC is that a cache must self-invalidate after a certain number of reads, specifically once an access counter reaches a predefined limit. In our model, we abstract the access counter by just having two rules corresponding to a Read in the Shared state: one where the access count is within its limit and another for when the limit has been reached. There is thus a non-deterministic choice between the two options:

$$c[a].\text{state} = S \implies \text{SendGetS}(c, Dir, a);$$
$$c[a].\text{state} := WS$$

$$\text{(Read S[MAX])}$$

$$c[a].\text{state} = S \implies //\text{do nothing}$$

$$\text{(Read S[<MAX])}$$

In the first rule, the access count has been reached, causing self-invalidation followed by re-requesting a fresh value from the directory; in the second rule, the access count has not been reached, and the cache is free to read hit on its own value with no further action. The model checker accounts for the non-deterministic choice between these rules.

Once the full ruleset in the model checker is defined, the rules are then exhaustively applied using an appropriate strategy (e.g. breadth first, depth first) until every possible state of the model has been enumerated; during this process of state enumeration the model checker checks that it can always proceed to another state (deadlock freedom) and that any defined invariants hold for each state. For efficiency, Murφ also reduces the set of states which need to be enumerated by using various techniques, such as symmetry reduction [25].

The next problem was to decide what property to check the model against. The derived properties which usually hold for CCPs, like SWMR do not hold for TSO-CC, by design, so a new strategy has to be applied. Our verification strategy is to establish that TSO-CC satisfies TSO by: (a) devising TSO-LB, a finite operational model for abstracting TSO-CC, (b) proving that TSO-LB shows only TSO behaviour, and (c) showing that TSO-CC is a refinement of TSO-LB within a model checker.

B. TSO-LB operational model

This section introduces the abstract TSO load-buffering model (TSO-LB). For our approach, existing operational models of TSO [19], [24] are not ideal for two reasons. The first being that they require unbounded buffers, making algorithmic verification difficult. Second, a refinement between a lazy CCP

and an existing store-buffering model would be difficult, as a lazy CCP effectively follows a load-buffering rather than a store-buffering approach: loads, viz. reads, hitting on a locally "buffered" (potentially) stale value, until the current value is pulled in (i.e. propagates) from global memory via a self-invalidate. The load-buffering based operational model formalised below abstracts a lazy CCP better and hence simplifies verification.

Definition 1 (Labelled Transition System): A *labelled transition system* (LTS) is a tuple $(\mathcal{L}, \mathcal{Q}, \mathcal{I}, \mathcal{T})$ where, $\mathcal{L}$ is a set of *labels*, $\mathcal{Q}$ is a set of *states*, $\mathcal{I} \subseteq \mathcal{Q}$ is a set of initial states and $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{L} \times \mathcal{Q}$ is the *transition relation*.

If $(q, l, q') \in \mathcal{T}$ then we say there is a transition labelled $l \in \mathcal{L}$ from state $q \in \mathcal{Q}$ to state $q' \in \mathcal{Q}$ and we may abbreviate this as $q \xrightarrow{l} q'$.

Definition 2 (TSO-LB): We define an LTS for TSO-LB as follows. The transition relation is given by the rules:

$$\frac{\text{local}_q(p)(a) = v}{q \xrightarrow{\text{Read}(p,a,v)} q} \text{ READ}$$

$$\frac{}{q \xrightarrow{\text{Write}(p,a,v)} \langle \text{local}_q[(p)(a) \mapsto v], \text{global}_q[(a) \mapsto v] \rangle} \text{ WRITE}$$

$$\frac{}{q \xrightarrow{\tau} \langle \text{local}_q[(p) \mapsto \text{global}_q], \text{global}_q \rangle} \text{ PROPAGATE}$$

where P is a finite set of processors, with $p \in P$; A is a finite set of addresses (memory locations), with $a \in A$; V is a finite set of data values, with $v \in V$; $\text{local}_q : P \to A \to V$ is a function where $\text{local}_q(p)(a)$ is the value at address a in the local buffer of p in state q and $\text{global}_q : A \to V$ is a function where $\text{global}_q(a)$ is the value at address a in the global buffer in state q.

The set of states $\mathcal{Q}$ consists of all pairs $\langle \text{local}_q, \text{global}_q \rangle$ and the set of labels $\mathcal{L} = \{\tau, \text{Read}(p, a, v), \text{Write}(p, a, v)\}$ where $p \in P$, $a \in A$, $v \in V$ and τ is the silent action. We define the set of initial states $\mathcal{I}$ to be $\mathcal{I} \triangleq \{q : \forall p \in P. \forall a \in A. \text{local}_q(p)(a) = \text{global}_q(a)\}$.

C. TSO-LB satisfies TSO

In the following, we outline a proof sketch that the TSO-LB operational model defined in Definition 2 permits only TSO behaviour. Since TSO-LB is defined as a LTS, its behaviour is defined with respect to an arbitrary trace of this LTS. We show (Theorem 1), by means of an interpretation of logical and physical time over these traces, that the behaviour satisfies the `herd` axiomatic characterisation of TSO [26]. We also show via a counterexample that TSO-LB does not permit all allowable behaviours of TSO, i.e. TSO-LB is in fact stricter than TSO.

Theorem 1 (TSO-LB satisfies TSO): The read and write events of traces of the TSO-LB LTS satisfy the TSO axiomatic MCM (as formalised in Alglave et al. [26].

Our proof strategy starts with defining a trace P of TSO-LB (Definition 3). The trace order might be seen as the physical-time representation of events, which contains writes, reads, and propagates. We will then construct a *strict linear order L* from P which contains the same writes and reads (with the same values). We then show how to instantiate the required

ordering relations from the `herd` framework of Alglave et al. [26] from L, and show all those orders are contained in L. This will then allow us to show that the `herd` axiomatic constraints of TSO hold over the write and read events. Note that we assume a simplified TSO model excluding fences, as TSO-LB does not model fences by definition.

Definition 3 (Trace): A *trace* of an LTS is a sequence (finite or infinite) of labels that results from a path of transitions starting at the initial state. Let us call this trace order P.

Definition 4 (Logical-time L): We define L to be an order on the read and write events in the trace P. All writes $\mathsf{Write}(p, a, v)$ appear in L in the same order as in the physical-time trace P. A read $\mathsf{Read}(p, a, v)$ is pulled backwards in the trace to just after the event in P which made the processor p get the value v for a. Such an event is either a write from the same processor p, or a propagate to the processor p (if $\mathsf{Read}(p, a, v)$ reads from a write on another processor).

Note that several reads from the same processor can be pulled back to the same point in this scheme, if the same address is read by multiple reads, or if the propagated values for different addresses for the same propagate event are read from by different reads. In such a case, we order these multiple reads in L (which have to be from the same processor) according to program order.

Definition 5 (`co` in TSO-LB): The order `co` is defined in TSO-LB as $\mathsf{Write}(p, a, v) \xrightarrow{\mathsf{co}} \mathsf{Write}(q, b, w)$ if and only if $\mathsf{Write}(p, a, v)$ occurs before $\mathsf{Write}(q, b, w)$ in the physical-time trace P, and the addresses a and b are the same. Note that p and q may be the same or different processors, and v and w the same or different values.

Definition 6 (`rf` in TSO-LB): The order `rf` is defined as $\mathsf{Write}(p, a, v) \xrightarrow{\mathsf{rf}} \mathsf{Read}(q, a, v)$ where p and q may be the same or different processors, and the read gets its value from the write.

We can now show that `co`, `rf`, and all the derived relations of the `herd` TSO formalisation are sub-orders of L. Then all axioms state the acyclicity and irreflexivities of various order relations, which are satisfied by any sub-orders of a strict linear order L. For the complete proof we refer to the online appendix.[2]

D. Weak simulation by model checking

Our core goal here is to check that a value read from a memory location by a processor at any point in time adheres to the TSO-LB specification, if all memory accesses are governed by the TSO-CC protocol.

We model both the TSO-LB specification and the TSO-CC protocol as labelled transition systems. In both cases, the labels are either *observable actions* concerning reads and writes or they are *silent actions*. For convenience below, we use the single label τ for all silent actions, though in our implementation it is useful to consider each system having a number of silent actions.

Our formal notion of correctness is that every observable trace of the TSO-CC protocol LTS is also an observable trace

---

[2] https://github.com/icsa-caps/tso-cc/blob/master/TSO-LB-proof.pdf

of the TSO-LB specification LTS. An *observable trace* is a trace with all the silent actions removed. We establish this inclusion property of observable traces by exhibiting a weak simulation relation between the TSO-CC LTS and the TSO-LB LTS such that the pair of initial states of the two LTSs is included in the relation.

A *weak simulation relation* shows step-by-step that for every observable action in TSO-CC there is a corresponding observable action in TSO-LB; it makes no attempt to match the silent actions in the two LTSs. This notion of weak simulation may be defined more formally as follows (following Milner [27]).

Definition 7 (Weak transition): Let $\mathcal{A} = (\mathcal{L}, \mathcal{Q}, \mathcal{I}, \mathcal{T})$ be an LTS. A *weak transition* $q \xRightarrow{l} q'$ is defined as $q \xrightarrow{\tau}{}^* x \xrightarrow{l} y \xrightarrow{\tau}{}^* q'$ for some x, y, where $\xrightarrow{\tau}{}^*$ is the reflexive transitive closure of $\xrightarrow{\tau}$ and $q, q', x, y \in \mathcal{Q}$, $l \in \mathcal{L}$ and $l \neq \tau$.

Later we use the notation $q \implies q'$ for $q \xrightarrow{\tau}{}^* q'$ or, if we allow multiple silent-action labels, to say that q' can be reached from q by zero or more transitions labelled by silent actions.

Definition 8 (Weak simulation): Let $\mathcal{C} = (\mathcal{L}, \mathcal{Q}_C, \mathcal{I}_C, \mathcal{T}_C)$ and $\mathcal{A} = (\mathcal{L}, \mathcal{Q}_A, \mathcal{I}_A, \mathcal{T}_A)$ be two LTSs with the same label set. Let $l \in \mathcal{L}$ be an observable action. A *weak simulation* $\mathcal{W} \subseteq \mathcal{Q}_C \times \mathcal{Q}_A$ is a binary relation such that if $(p, q) \in \mathcal{W}$, written $p\mathcal{W}q$, then

1) if $p \xrightarrow{l} p'$ then there exists $q' \in \mathcal{Q}_A$ such that $q \xRightarrow{l} q'$ and $p'\mathcal{W}q'$, and
2) if $p \xrightarrow{\tau} p'$ then there exists $q' \in \mathcal{Q}_A$ such that $q \implies q'$ and $p'\mathcal{W}q'$.

In our setting, $\mathcal{Q}_C$ are the states of the CCP and $\mathcal{Q}_A$ are the states of the MCM.

To prove that there exists a weak simulation relation using a model checker, we construct an unlabelled transition system $\mathcal{M} = (\mathcal{Q}, \mathcal{T})$ from the two LTSs with $\mathcal{Q} = \mathcal{Q}_C \times \mathcal{Q}_A$ and a specially crafted transition relation $\mathcal{T}$. If a certain property holds for every reachable state of $\mathcal{M}$, then the set of reachable states is a weak simulation relation between $\mathcal{C}$ and $\mathcal{A}$. As the initial state of $\mathcal{M}$ is a pair of the initial states of $\mathcal{C}$ and $\mathcal{A}$, we have that the initial state pair are related by the weak simulation, and hence every observable trace of $\mathcal{C}$ is also an observable trace of $\mathcal{A}$. We can describe the transition relation and checked property as follows. The transition relation $\langle p, q \rangle \longrightarrow \langle p', q' \rangle$ is defined as $\exists l \in L. p \xrightarrow{l} p' \wedge q' = \mathrm{last}(\mathrm{AbsWitness}(p, q, l))$ where $\mathrm{AbsWitness}(p, q, l)$ computes an alternating sequence of abstract states and labels $\langle q_0, l_0, q_1, l_1, \ldots, q_n \rangle$ for some $n \geq 0$, $q = q_0$ and the $\mathrm{last}()$ function picks out the last state q_n of such a sequence.

The checked property $\mathrm{Match}(\langle p, q \rangle)$ is defined as $\forall l \in L, p' \in \mathcal{Q}_C. p \xrightarrow{l} p' \Rightarrow \mathrm{AbsWitness}(p, q, l)$ is a witness for:

1) $q \xRightarrow{l} \mathrm{last}(\mathrm{AbsWitness}(p, q, l))$ if l is observable. and
2) $q \implies \mathrm{last}(\mathrm{AbsWitness}(p, q, l))$ if $l = \tau$.

Here, an alternating sequence of abstract states and labels $\langle q_0, l_0, q_1, l_1, \ldots, q_n \rangle$ *is a witness for* $q_0 \implies q_n$ if all the l_i are silent and $q_i \xrightarrow{l_i} q_{i+1}$ for all $i \in \{0, \ldots, n-1\}$, and *is a witness for* $q_0 \xRightarrow{l} q_n$ if there exists a unique $l_i = l$ in

the sequence such that $\forall j \neq i$ l_j is silent and $q_i \xrightarrow{l_i} q_{i+1}$ for all $i \in \{0, \ldots, n-1\}$. Witnesses for weak transition instances enable the straightforward checking of the truth of instances.

A conceptual sketch of the witness function $\mathrm{AbsWitness}$ we use is as follows:

- If TSO-CC does a write action, then TSO-LB is made to take a single corresponding write action step.
- For silent transitions of TSO-CC, the witness is a single state—i.e. TSO-LB takes no steps.
- If TSO-CC does a read action, then in TSO-LB we either do a read action, or a propagate action followed by a read. We settle for the single read step if it is allowed by TSO-LB. If not, we go for the 2 step witness. As propagate is the only silent action in TSO-LB and it is idempotent, there are no other options to consider.

In general the abstract LTS might permit several silent transitions and the $\mathrm{AbsWitness}$ function has to embody some strategy for testing possible silent actions; however, it is worthy of note that the trivial strategy, as described here, is generally applicable to checking *any* CCP against TSO-LB.

E. Weak simulation in Murφ

To realise the above in Murφ we started with the TSO-CC Murφ model introduced in Section III-A and augmented the state with components for the TSO-LB specification. At the rules in the TSO-CC model where observable actions (reads/writes) are performed, we also step forward the TSO-LB model with the same actions, as explained above.

Coding the Match predicate is much simpler than the conceptual presentation above suggests. For the step forward of the TSO-LB system on write actions, the step is guaranteed by construction to satisfy the TSO-LB labelled transition relation, there is nothing to check. Only for the read action do we need to check that the value read in the TSO-LB specification actually matches that from the TSO-CC system. We simply use an invariant in Murφ to check the read value at each read step.

In the following, we detail how we implemented the transition system model and Match check in Murφ. The implementation of TSO-LB involves a pair of arrays to represent the global and local buffers for each cache and address, Murφ procedures $\mathrm{TSOStore}$ and $\mathrm{TSOUpdate}$, and the Murφ function $\mathrm{TSOVerify}$. These functions compute the next TSO-LB state for the Write, Propagate, and Read TSO-LB rules respectively. In addition $\mathrm{TSOVerify}$ returns a Boolean value indicating whether TSO-LB can indeed make one or two steps forward that result in a correct read. Calling $\mathrm{TSOVerify}$ returns true if the expected value is in the local buffer, or it tries a $\mathrm{TSOUpdate}$ and returns true if the expected value is now in the local buffer, else it returns false. A Murφ invariant ensures that $\mathrm{TSOVerify}$ always returns true.

The rules of TSO-CC incorporate these TSO-LB procedures and function. Taking our previous example rules, we amend them as follows:
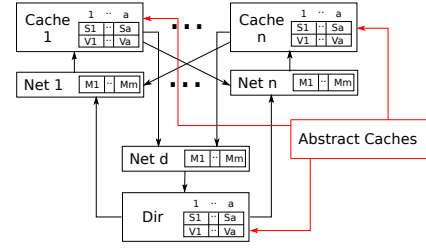

Fig. 2. Parameterised model structure with abstract caches.

$$c[a].\mathtt{state} = \mathtt{E} \implies c[a].\mathtt{val} := v$$
$$c[a].\mathtt{state} := M;$$
$$\boxed{\mathtt{TSOStore}(c, a, v)}$$

(Write E)

$$c[a].\mathtt{state} = \mathtt{S} \implies \text{//do nothing;}$$
$$\boxed{\mathtt{Assert}(\mathtt{TSOVerify}(c, a, c[a].\mathtt{val}))}$$

(Read S[<MAX])

and likewise wherever a Read or Write action occurs in the CCP model. In this way our model shows that the values at the CCP level are consistent with the values at the MCM level.

Thus, for a fixed number of processors, we show that the simulation relation between TSO-CC and TSO-LB holds. The next problem was to show that the simulation relation holds for any number of processors. The next section shows how we solved this problem.

IV. TSO-CC with n processors satisfies TSO

After showing that TSO-CC indeed satisfies TSO for a finite number of processors, we now show that this is also the case irrespective of the number of processors. In this section we present a parameterised model, parameterised in the number of processors, showing the same weak simulation relation between TSO-CC and TSO still applies with n processors.

In order to define a parameterised model we follow the method of Chou et al. [9], who in turn refined the ideas of McMillan [28]; the method is proven mathematically correct by Krstić [29]. The essence of the method is that one takes the original concrete model, but adds a new *abstract cache*. The abstract cache represents any number of caches connected to the concrete model (Figure 2). Initially the abstract cache can send any possible message to the concrete caches. This over-approximated set of messages is then reduced to only the set of legal messages by a process of counterexample guided abstraction refinement [30].

The initial over-approximated set of messages coming from the abstract cache will generate a counterexample when a spurious message is sent. One can then introduce a restriction to the abstract cache which disallows the spurious message. However it is then necessary to show that the restriction is valid and does not lead to an under-approximation of legal

messages. In order to achieve this, one can write a *non-interference lemma* which shows the restricted message cannot occur in the concrete model. The key to the method is that the process is manual, but simply mechanical: the restriction is guided by the counterexample, the lemma is guided by the restriction, then the model checker checks both simultaneously and automatically. The apparent circular reasoning in proving the lemma on the amended system is dealt with by example by Chou et al. [9] and proved correct by Krstić [29].

A. Parameterised model in Murφ

The implementation of the parameterised model begins with the definition of a set of rules which generate all the possible messages which could be received at each node from the abstract caches. For example, one of the rules which handles the receipt of a `DataX` message at a cache is:

$$c[a].\texttt{state} = \texttt{WX}$$
$$\Longrightarrow \texttt{SendAck}(c, Dir, a);$$
$$c[a].\texttt{state} := \texttt{M};$$
$$\texttt{InvalidateAllOtherLines}(c, a)$$
$$\text{(Cache Recv DataX Abs)}$$

and similar rules are defined for each combination of node, state, and message received.

To extend the MCM to the parameterised model, we must consider what happens when our observable actions are performed by the abstract part of the parameterised model. As we do not track the state of abstract processors a Read action is not explicitly defined in the abstract part of the model, however a Write action by an abstract processor would have an effect (eventually) on the state of the concrete caches. We do not keep track of values at the abstract caches, so local buffers for abstract caches are not needed in the memory model. However, a Write at an abstract cache will go to the global buffer, because it may at some point be read by a concrete cache.

Thus, we implement a new function, `TSOStoreAbs`, which writes only to the global buffer. Now, in our abstract cache rules, we add a call to `TSOStoreAbs` wherever we see a Write action. For example, when the directory receives a Data message from an abstract cache a Write has occurred and we record this in the global buffer:

$$c[a].\texttt{state} = \texttt{WS} \Longrightarrow dir[a].\texttt{val} := v;$$
$$\texttt{TSOStoreAbs}(a, v)$$
$$dir[a].\texttt{state} := \texttt{S}$$
$$\text{(Dir Recv Data Abs)}$$

Once these rules are defined the model checker will generate all possible messages coming from the abstract cache. At this stage we have an *over-approximation* of the system. Of course, some of these messages will not be valid in the current state. When this occurs a counterexample will be generated by the model checker. The modeller must then inspect the counterexample and work out why the message was spurious.

It is then possible to add a restriction to the rule that generated it such that the spurious message is eliminated.

For example, in the above rule (Cache Recv DataX Abs), we allow the cache to receive a `DataX` even when it is not the owner of the cache line. This produces a counterexample, because to receive a `DataX` from another cache (here an abstract cache) the other cache must have received a `FwdX` message first telling it to forward data to the new owner. Therefore the receiving cache must be the owner. To eradicate the counterexample we must add a restriction to check the receiving cache is the owner:

$$c[a].\texttt{state} = \texttt{WX} \boxed{\wedge \texttt{ IsOwner}(c, a)}$$
$$\Longrightarrow \texttt{SendAck}(c, Dir, a);$$
$$c[a].\texttt{state} := \texttt{M};$$
$$\texttt{InvalidateAllOtherLines}(c, a)$$
$$\text{(Cache Recv DataX Abs)}$$

However, we must now show that the restriction is not too strict, i.e. we have not inadvertently caused the system to be *under-approximated* and, in essence, we are not changing the protocol. To do this, we introduce a *non-interference lemma*; this is a lemma which states the restriction as an invariant in the context of the concrete model, thus ensuring that the spurious messages eliminated are indeed not possible in the fully concrete model. For example, the lemma for the above restriction is:

$$\forall n \forall a \forall i.\ net[n][a][i].\texttt{msgType} = \texttt{DataX} \Longrightarrow \texttt{IsOwner}(n, a)$$

where n is a node, a is an address, and i is a position in the message buffer. This is implemented in the model checker as an invariant[3] and if it does not fail then we know that we have not over-constrained the abstract cache.

Now, the model checker may catch a new counterexample. If this is the case then we repeat the process until all counterexamples are eliminated. Once all counterexamples are eliminated, we are done.

V. RESULTS

In summary, the result of applying the method described in this paper to the TSO-CC protocol was that we showed that the protocol does, in fact, conform to the TSO memory model with any number of processors. Execution times for checking the full model are in the order of 14–15 hours on a single core of an Intel Xeon 1.8GHz machine with 64GB of RAM. The process of manually refining the model for parameterisation required 30 passes around the refinement loop, generating 30 non-interference lemmas. The time needed to define each lemma varied, depending on the complexity of the counterexample—at this stage, detailed knowledge of the protocol was a boon. Of note, however, is that for each pass around the refinement loop does not require 14 hours of model checking; generally, the model checker needed only to run

---

[3]Details of the restrictions and non-interference lemmas can be found in the model source at https://github.com/icsa-caps/tso-cc

for a few minutes to find the next counterexample—this time gradually increased as more counterexamples were eliminated.

It is also worth noting that one needs to consider vacuity in model checking and we must consider whether or not the specification holds trivially. We believe that our model is not vacuous, given that during the development of the model we observed and analysed a number of counter-example traces, both when we intentionally introduced bugs, during development, and when we iterated through the CEGAR loop, generating and refining the non-interference lemmas.

VI. CONCLUSION

We have shown that it is possible to verify a modern, lazy CCP against its counterpart TSO MCM. Our main contributions have been three fold:

1) the extension of a previous method [10] in order to formally verify a lazy CCP against the TSO weak MCM that it implements: the key novelty that enables this extension is the introduction of the new abstract operational model, TSO-LB;

2) a proof that our TSO-LB model satisfies a well-regarded axiomatic model of TSO;

3) extending the result of 1) to an arbitrary number of processors, using the parameterisation method of Chou et al. [9]: in establishing this result, we demonstrate that Chou et al.'s method for parameterised verification can be used to prove that a CCP refines an abstract operational model, not just for verifying protocol-specific properties such as SWMR.

We believe it would be straightforward to use our approach to verify other lazy CCPs that implement TSO. As such, one direction of future work is to improve the degree of automation in the method. Whilst the process of parameterisation of the model is simple, requiring more knowledge of the protocol than of formal methods, of note is the time and effort required to write restrictions, write lemmas, model check, and repeat. It is our belief that more of this process may be automated, as was the goal of both Chou et al. [9] and Krstić [29]. Some research on this topic already exists in the literature, for example O'Leary et al. [23] and Bingham et al. [31]. Another direction for future work is to check how we might use results similar to those of Henzinger et al. [32] to justify the verification for arbitrary numbers of addresses and data values.

REFERENCES

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[2] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[3] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," *PACT*, pp. 155–166, 2011.

[4] A. Ros, "Complexity-Effective Multicore Coherence," *ACM PACT*, pp. 241–251, 2012.

[5] T. J. Ashby, P. Díaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Transactions on Computers*, vol. 60, no. 4, pp. 472–483, 2011.

[6] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2014, pp. 165–176.

[7] D. Abts, S. Scott, and D. J. Lilja, "So many states, so little time: Verifying memory coherence in the Cray X1," in *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2003*, 2003.

[8] R. Komuravelli, S. V. Adve, and C.-T. Chou, "Revisiting the Complexity of Hardware Cache Coherence and Some Implications," *ACM TACO*, vol. 11, no. 4, pp. 1–22, 2014.

[9] C. Chou, P. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," *FMCAD*, pp. 382–398, 2004.

[10] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan, "Shared Memory Consistency Protocol Verification Against Weak Memory Models: Refinement via Model-Checking," *CAV*, pp. 121–138—-, 2002.

[11] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: using hb graphs to verify the coherence-consistency interface," in *MICRO'15*, 2015, pp. 26–37.

[12] K. L. McMillan, "Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking," *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pp. 179–195, 2001.

[13] X. Chen, Y. Yang, G. Gopalakrishnan, and C. T. Chou, "Reducing verification complexity of a multicore coherence protocol using assume/guarantee," *FMCAD 2006*, vol. 3, pp. 81–88, 2006.

[14] F. Pong and M. Dubois, "Formal Verification Of Complex Coherence Protocols Using Symbolic State Models," *Journal of the ACM*, vol. 45, no. 4, pp. 557–587, 1998.

[15] ——, "Formal automatic verification of cache coherence in multiprocessors with relaxed memory models," *IEEE TPDS*, vol. 11, no. 9, pp. 989–1006, 2000.

[16] P. A. Abdulla, M. F. Atig, A. Bouajjani, and T. P. Ngo, "The Benefits of Duality in Verifying Concurrent Programs under TSO," *27th International Conference on Concurrency Theory (CONCUR 2016)*, vol. 59, no. 5, pp. 1–5, 2016.

[17] M. Elver, "Memory Consistency Directed Cache Coherence Protocols for Scalable Multiprocessors," Ph.D. dissertation, University of Edinburgh, 2016.

[18] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running tests against hardware," *Lecture Notes in Computer Science*, vol. 6605 LNCS, pp. 41–44, 2011.

[19] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: X86-TSO," in *Lecture Notes in Computer Science*, vol. 5674 LNCS, 2009, pp. 391–407.

[20] M. Elver and V. Nagarajan, "McVerSi: A test generation framework for fast memory consistency verification in simulation," in *HPCA*, vol. 2016-April, 2016, pp. 618–630.

[21] D. L. Dill, "The Murphi Verification System," in *CAV 96: Computer-Aided Verification*, vol. 1102, 1996, pp. 390–393.

[22] S. Burckhardt, R. Alur, and M. M. Martin, "Verifying safety of a token coherence implementation by parametric compositional refinement," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2005, pp. 130–145.

[23] J. O'Leary, M. Talupur, and M. R. Tuttle, "Protocol verification using flows: An industrial experience," in *9th International Conference Formal Methods in Computer Aided Design, FMCAD 2009*, 2009, pp. 172–179.

[24] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO," *Communications of the ACM*, vol. 53, no. 7, p. 89, 2010.

[25] C. Norris IP and D. L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, no. 1-2, pp. 41–75, 1996.

[26] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats," *ACM TOPLAS*, vol. 36, no. 2, pp. 1–74, jul 2014.

[27] R. Milner, *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, 1999.

[28] K. L. McMillan, "Verification of infinite state systems by compositional model checking," *Lecture Notes in Computer Science*, vol. 1703, pp. 219–237, 1999.

[29] S. Krstić, "Parametrized System Verification with Guard Strengthening and Parameter Abstraction," *Electronic Notes in Theoretical Computer Science*, pp. 1–13, 2005.

[30] E. Clarke, "Counterexample-guided abstraction refinement," *Proceedings of the International Workshop on Temporal Representation and Reasoning*, pp. 7–8, 2003.

[31] J. Bingham, "Automatic non-interference lemmas for parameterized model checking," *FMCAD*, pp. 1–8, 2008.

[32] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Verifying sequential consistency on shared-memory multiprocessor systems," in *CAV*, vol. 1633, 1999, pp. 301–315.