A Proof of Correctness for the Tardis Cache Coherence Protocol

Xiangyao Yu, Muralidaran Vijayaraghavan, Srinivas Devadas

{yxy, vmurali, devadas}@mit.edu

Massachusetts Institute of Technology

Abstract

We prove the correctness of a recently-proposed cache coherence protocol, Tardis, which is simple, yet scalable to high processor counts, because it only requires $O(\log N)$ storage per cacheline for an N-processor system. We prove that Tardis follows the sequential consistency model and is both deadlock- and livelock-free. Our proof is based on simple and intuitive invariants of the system and thus applies to any system scale and many variants of Tardis.

1 Introduction

Tardis [37] is a recently proposed cache coherence protocol that is able to scale to a large number of cores. Unlike full-map directory protocols [7, 35], Tardis does not keep the $O(N)$ (N is the core count) sharer information for each cacheline. In Tardis, only the owner ID of each cacheline ($O(\log N)$) and logical timestamps ($O(1)$) for each cacheline are maintained. Unlike the snoopy bus coherence protocol [15], or limited directory protocols such as ACKwise [20], Tardis does not broadcast messages to maintain coherence.

In Tardis, each load or store is assigned a logical timestamp which may not agree with the physical time. The global memory order then simply becomes the timestamp order which is explicit in the protocol. This makes it much simpler to reason about the correctness of Tardis. Despite its simplicity, however, no proof of correctness has yet been published. We provide a simple and straightforward proof in Section 3; our proof is simpler than existing proofs for snoopy and directory protocols such as [32, 36].

In this paper, we formally prove the correctness of the Tardis protocol by showing that an execution of a program using Tardis strictly follows *Sequential Consistency* (SC). We also prove that the Tardis protocol can never deadlock or livelock.

The original Tardis protocol [37] was designed for a shared memory multicore processor. A number of optimization techniques were applied for performance improvement. These optimizations, however, may not be desirable in other kinds of shared memory systems. Therefore, in this paper we first extract the core algorithm of Tardis and prove its correctness. We then focus on correctness of generalizations of the base protocol.

We prove the correctness of Tardis by developing simple and intuitive system invariants. Compared to the popular model checking [12, 18, 28] verification techniques, our proof technique is able to scale to high processor counts. More important, the invariants we developed are more intuitive to system designers and thus provide more guidance for system implementation.

The rest of the paper is organized as follows. The Tardis protocol is formally defined in Section 2. It is proven to obey sequential consistency in Section 3 and to be deadlock-free and livelock-free in Section 4. In Section 5, we generalize the proofs to systems with main memory. Section 6 describes related work and Section 7 concludes the paper.

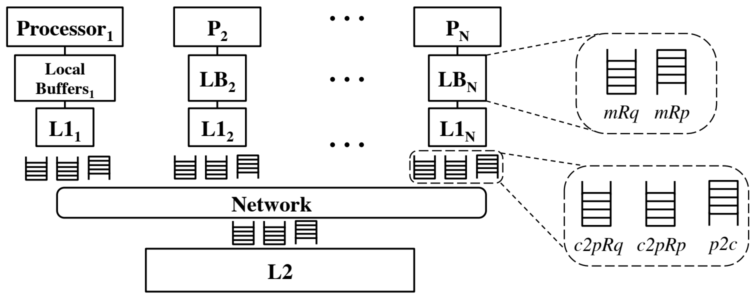

Figure 1: Architecture of a shared memory multicore processor.

2 Tardis Coherence Protocol

We first present the model of the shared memory system we use, along with our assumptions, in Section 2.1. Then, we introduce system components of the Tardis protocol in Section 2.2 and formally specify the protocol in Section 2.3.

2.1 System Description

Fig. 1 shows the architecture of a shared memory system based on which Tardis will be defined. The processors can execute instructions in-order or out-of-order but always commit instructions in order. A processor talks to the memory subsystem through a pair of *local buffers* (LB). Load and store requests are inserted into the *memory request buffer* (*mRq*) and responses from the memory subsystem are in the *memory response buffer* (*mRp*).

We model a two-level memory subsystem with all the data fitting into the L2 caches. The network between L1 and L2 caches is modeled as buffers. *c2pRq* contains requests from L1 (child) to L2 (parent), *c2pRp* contains responses from L1 to L2, and *p2c* contains messages (both requests and responses) from L2 to L1. For simplicity, all the buffers are modeled as FIFOs and *get_msg*() returns the head message in the buffer. However, the protocol also works if the buffers only have the FIFO property for the same address. Each L1 cache has a unique *id* from 1 to N and each associated buffer has the same *id*. An L1 cacheline or a message in a buffer has the same *id* as the L1 cache or the buffer it is in.

2.2 System Components in Tardis

The Tardis protocol is built around the concept of logical timestamps. Each memory operation in Tardis has a corresponding timestamp which indicates its global memory order. The memory dependency is expressed using timestamps and no sharer information is maintained. For simplicity, we assume the timestamps to be large enough that they never overflow (e.g., 64-bits). Timestamp compression algorithms are able to achieve much smaller timestamps (e.g., 16-bits) in practice [37].

At a high level, if a load observes the value of a previous store, then the load should be ordered after the store in the logical time (and thus global memory order). Similarly, a store should be ordered after a load if the load does not observe the stored value. To keep track of the timestamps of each operation, every cacheline in Tardis has a read timestamp (*rts*) and a write timestamp (*wts*). The *wts* is the timestamp of the last store and *rts* is the timestamp of the last (potential) load to the cacheline ($wts \leq rts$). Similar to a directory protocol, a cacheline can be cached in L1 in either M or S states. Only one L1 can obtain the M state at any time to modify the cacheline, and multiple L1s can share the line in the S state. Timestamps are also required for messages in the buffers. Table 1 summarizes the format of caches and buffers modeled in the system. The differences between Tardis and a directory protocol are highlighted in red.

An L1 cacheline contains five fields: *state*, *data*, *busy*, *wts* and *rts*. The *state* can be M, S or I. For ease of discussion, we define a total ordering among the three states $I < S < M$. A cacheline has *busy* = *True* if a request to L2 is outstanding. This prevents duplicated requests. An L2 cacheline contains one more field

Table 1: System Components

Component	Format	Message Types
$L1$	$L1[addr] = (state, data, busy, \mathbf{wts}, \mathbf{rts})$	-
$L2$	$L2[addr] = (state, data, busy, owner, \mathbf{wts}, \mathbf{rts})$	-
mRq	$mRq.entry = (type, addr, data, \mathbf{pts})$	$LdRq$, $StRq$
mRp	$mRp.entry = (type, addr, data, \mathbf{pts})$	$LdRp$, $StRp$
$c2pRq$	$c2pRq.entry = (id, type, addr, \mathbf{pts})$	$GetS$, $GetM$
$c2pRp$	$c2pRp.entry = (id, addr, data, \mathbf{wts}, \mathbf{rts})$	$WBRp$
$p2c$	$p2c.entry = (id, msg, addr, state, data, \mathbf{wts}, \mathbf{rts})$	ToS, ToM, $WBRq$

Table 2: State Transition Rules for L1.

Rules and Condition	**Action**
LoadHit **let** $(type, addr, \_, \mathbf{pts}) = mRq.get\_msg()$ **let** $(state, data, busy, \mathbf{wts}, \mathbf{rts}) = L1[addr]$ **condition:** $\neg\, busy \wedge type = S \wedge \mathbf{(state \geq S \wedge pts \leq rts)}$	mRq.deq() mRp.enq$(type, addr, data, \mathbf{max(pts, wts)})$ **If** $(state = M)$ $\quad \mathbf{rts := max(pts, rts)}$
StoreHit **let** $(type, addr, data, \mathbf{pts}) = mRq.get\_msg()$ **let** $(state, data, busy, \mathbf{wts}, \mathbf{rts}) = L1[addr]$ **condition:** $\neg\, busy \wedge type = state = M$	$\mathbf{let\ pts' = max(pts, rts + 1)}$ mRq.deq() mRp.enq$(type, addr, _, \mathbf{pts'})$ $\mathbf{wts := pts'}$ $\mathbf{rts := pts'}$
L1Miss **let** $(type, addr, data, \mathbf{pts}) = mRq.get\_msg()$ **let** $(state, data, busy, \mathbf{wts}, \mathbf{rts}) = L1[addr]$ **condition:** $\neg\, busy \wedge (state < type \vee \mathbf{state = type = S \wedge pts > rts})$	$c2pRq$.enq$(id, type, addr, \mathbf{pts})$ $busy := True$
L2Resp **let** $(id, msg, addr, state, data, \mathbf{wts}, \mathbf{rts}) = p2c.get\_msg()$ **let** $(l1state, l1data, busy, \mathbf{l1wts}, \mathbf{l1rts}) = L1[addr]$ **condition:** $msg = Resp$	$p2c$.deq() $l1state := state$ $l1data := data$ $busy := False$ $\mathbf{l1wts := wts}$ $\mathbf{l1rts := rts}$
Downgrade **let** $(state, data, busy, \mathbf{wts}, \mathbf{rts}) = L1[addr]$ **condition:** $\neg\, busy \wedge \exists\, state'.\ state' < state$ $\wedge$ LoadHit and StoreHit cannot fire	**If** $(state = M)$ $\quad c2pRp$.enq$(id, addr, data, \mathbf{wts}, \mathbf{rts})$ $state := state'$
WriteBackReq **let** $(state, data, busy, \mathbf{wts}, \mathbf{rts}) = L1[addr]$ **condition:** $p2c.get\_msg().msg = Req$ $\wedge$ LoadHit and StoreHit cannot fire	$p2c$.deq() **If** $(state = M)$ $\quad c2pRp$.enq$(id, addr, data, \mathbf{wts}, \mathbf{rts})$ $\quad state := S$

owner, which is the *id* of the L1 that exclusively owns the cacheline in the M state. As in L1, *busy* in L2 is set when a write back request (*WBRq*) to an L1 is outstanding.

Each entry in *mRq* contains four fields: *type*, *addr*, *data* and *pts*. The *type* can be S or M corresponding to a load or store request, respectively. The *pts* is a timestamp specified by the processor and the timestamp of the memory operation should be no less than *pts*. *mRp* has the same format as *mRq*, but *pts* here is the actual timestamp of the memory operation.

The format of the messages in the network buffers (*c2pRq*, *c2pRp* and *p2c*) is shown in Table 1, where the meaning of the fields are as in the cachelines or the messages in *mRq*. All network messages have a field *id* which is the *id* of the L1 cache that the message comes from or goes to. Messages in *p2c* have a field *msg*, which can be either *Req* or *Resp*; *p2c* may contain both requests and responses from L2 to L1 and *msg* differentiates the two types.

2.3 Protocol Specification

We now formally define the core algorithm of the Tardis protocol. The state transition rules for L1 and L2 caches are summarized in Table 2 and Table 3 respectively, with the differences between Tardis and a directory protocol highlighted in red. For all rules where a message is enqueued to a buffer, the rule can only fire if the buffer is not full.

Table 3: State Transition Rules for L2.

Rules and Condition	Action
ShReq_S **let** $(id,\ type,\ addr,\ pts) = c2pRq.get\_msg()$ **let** $(state,\ data,\ busy,\ owner,\ wts,\ rts) = L2[addr]$ **condition:** $type = S \wedge state = S$ $\wedge \exists\ pts'.\ pts' \geq rts \wedge pts' \geq pts$	$c2pRq.\mathrm{deq}()$ $rts := pts'$ $p2c.\mathrm{enq}(id,\ Resp,\ addr,\ S,\ data,\ wts,\ pts')$
ExReq_S **let** $(id,\ type,\ addr,\ pts) = c2pRq.get\_msg()$ **let** $(state,\ data,\ busy,\ owner,\ wts,\ rts) = L2[addr]$ **condition:** $type = M \wedge state = S$	$c2pRq.\mathrm{deq}()$ $state := M$ $owner := id$ $p2c.\mathrm{enq}(id,\ Resp,\ addr,\ M,\ data,\ wts,\ rts)$
Req_M **let** $(id,\ type,\ addr,\ pts) = c2pRq.get\_msg()$ **let** $(state,\ data,\ busy,\ owner,\ wts,\ rts) = L2[addr]$ **condition:** $state = M \wedge \neg\ busy$	$p2c.\mathrm{enq}(owner,\ Req,\ addr,\ \_,\ \_,\ \_,\ \_)$ $busy := True$
WriteBackResp **let** $(id,\ addr,\ data,\ wts,\ rts) = c2pRp.get\_msg()$ **let** $(state,\ l2data,\ busy,\ owner,\ l2wts,\ l2rts) = L2[addr]$	$c2pRp.\mathrm{deq}()$ $state := S$ $l2data := data$ $busy := False$ $l2wts := wts$ $l2rts := rts$

An important concept in Tardis is the lease. For a cacheline shared in an L1 cache, it also contains a lease which expires at the current *rts*. The data is only valid if the lease has not expired, i.e., *pts* from the request is less than or equal to *rts*. The *rts* in the L2 cache is the maximum of the *rts* of all the sharing L1 caches. When a shared cacheline in L2 cache gets a *GetM* request, it does not send invalidations as in a directory protocol, rather, it immediately returns exclusive ownership to the requesting processor, which will jump ahead in logical time and perform the store at $rts + 1$. If we consider logical time, the store happens after all the loads that do not observe its value, although in physical time, the store and the loads may happen simultaneously.

Specifically, the following six transition rules may fire in an L1 cache.

1. LoadHit. LoadHit can fire if the requested cacheline is in the *M* state or is in the *S* state and the lease has not expired. If it is in the *M* state, then *rts* is updated to reflect the latest load timestamp. The *pts* returned to the processor is no less than the cacheline's *wts*, which orders the load after the previous store in logical time.

2. StoreHit. StoreHit can only fire if the requested cacheline is in the *M* state in the L1 cache. Both *wts* and *rts* are updated to the timestamp of the store operation which is at least $rts + 1$. The store is thus logically ordered after all concurrent loads on the same address in other L1s.

3. L1Miss. If neither *LoadHit* nor *StoreHit* can fire for a request and the cacheline is not busy, it is an L1 miss and the request (*GetS* or *GetM*) is forwarded to the L2 cache. The cacheline is then marked as busy to prevent sending duplicated requests.

4. L2Resp. A response from L2 sets all the fields in the L1 cacheline. The *busy* flag is reset to *False* and the cacheline can serve the next request in the *mRq*.

5. Downgrade. A cacheline in the *M* or *S* states may downgrade if the cacheline is not busy and *LoadHit* and *StoreHit* cannot fire. For *M* to *S* or *I* downgrade, the cacheline should be written back to the L2 in a *WBRp* message. *S* to *I* downgrade, however, is silent, i.e., no message is sent.

6. WriteBackReq. When a cacheline in *M* state receives a write back request, the cacheline is returned to L2 in a *WBRp* message and the L1 state becomes *S*. If the request is to a cacheline in *S* or *I* state, the request is simply ignored. This corresponds to the case where the line self downgrades after the write back request (*WBRq*) is sent from the L2.

The following four rules may fire in the L2 cache.

7. ShReq_S. When a cacheline in the *S* state receives a shared request (i.e., *GetS*), both the *rts* and the returned *pts* are set to *pts′* which can be any timestamp greater than or equal to the current *rts* and *pts*. The *pts′* indicates the end of the lease for the cacheline. And the cacheline may be loaded at any logical time between *wts* and *pts′*. The returned message is a *ToS* message.

8. ExReq_S. When a cacheline in the S state receives an exclusive request (i.e., *GetM*), the cacheline is instantly returned in a *ToM* message. Unlike in a directory protocol, *no* invalidations are sent to the sharers. The sharing processors may still load their local copies of the cacheline and such loads have smaller timestamps than the store from the new owner processor.

9. Req_M. When a cacheline in the M state receives a request and is not busy, a write back request (i.e., *WBRq*) is sent to the current owner. *busy* is set to *True* to prevent sending duplicated *WBRq* requests.

10. WriteBackResp. Upon receiving a write back response (i.e., *WBRp*), data and timestamps are written to the L2 cacheline. The *state* becomes S and *busy* is reset to *False*.

3 Proof of Correctness

We now prove the correctness of the Tardis protocol specified in Section 2.3 by proving that it strictly follows sequential consistency. We first give the definition of sequential consistency in Section 3.1 and then introduce the basic lemma (Section 3.2) and timestamp lemmas (Section 3.3) that are used for the correctness proof.

Most of the lemmas and theorems in the rest of the paper are proven through induction. For each lemma or theorem, we first prove that it is true for the initial system state (base case) and then prove that it is still true after any possible state transition assuming that it was true before the transition.

In the initial system state, all the L1 cachelines are in the I state, all the L2 cachelines are in the S state and all the network buffers are empty. For all cachelines in L1 or L2, $wts = rts = 0$ and $busy = False$. Requests from the processors may exist in the *mRq* buffers. For ease of discussion, we assume that each initial value in L2 was set before the system starts at timestamp 0 through a store operation.

3.1 Sequential Consistency

According to Lamport [22], a parallel program is sequentially consistent if "*the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*". Using $<_m$ and $<_p$ to represent the global memory order and program order per processor respectively, sequential consistency (SC) is defined as follows [34].

Definition 1 (Sequential Consistency). *An execution of a program is sequentially consistent iff*

Rule 1: $\forall X, Y \in \{Ld, St\}$ *from the same processor,* $X <_p Y \Rightarrow X <_m Y$.

Rule 2: *Value of* $L(a)$ = *Value of* $Max_{<m}\{S(a)|S(a) <_m L(a)\}$, *where* $L(a)$ *and* $S(a)$ *are a load and a store to address* a*, respectively, and* $Max_{<m}$ *selects the most recent operation in the global memory order.*

In Tardis, the global memory order of sequential consistency is expressed using timestamps. Specifically, Theorem 1 states the invariants in Tardis that correspond to the two rules of sequential consistency. Here, we use $<_{ts}$ and $<_{pt}$ to represent (logical) timestamp order that is assigned by Tardis and physical time order that represents the order of events, respectively.

Theorem 1 (SC on Tardis). *An execution on Tardis has the following invariants.*

Invariant 1: *Value of* $L(a)$ = *Value of* $Max_{<ts}\{S(a)|S(a) \leq_{ts} L(a)\}$.

Invariant 2: $\forall S_1(a), S_2(a), S_1(a) \neq_{ts} S_2(a)$.

Invariant 3: $\forall S(a), L(a), S(a) =_{ts} L(a) \Rightarrow S(a) <_{pt} L(a)$.

Theorem 1 itself is not enough to guarantee sequential consistency; we also need the processor model described in Definition 2. The processor should commit instructions in the program order, which implies physical time order and monotonically increasing timestamp order. Both in-order and out-of-order processors fit this model.

Definition 2 (In-order Commit Processor). $\forall X, Y \in \{Ld, St\}$ *from the same processor,* $X <_p Y \Rightarrow X \leq_{ts} Y \land X <_{pt} Y$.

Now we prove that given Theorem 1 and our processor model, an execution obeys sequential consistency per Definition 1. We first introduce the following definition of the global memory order in Tardis.

Definition 3 (Global Memory Order in Tardis)**.**

$$X <_m Y \triangleq X <_{ts} Y \vee X =_{ts} Y \wedge X <_{pt} Y.$$

Theorem 2. *Tardis with in-order commit processors implements Sequential Consistency.*

Proof. According to Definitions 2 and 3, $X <_p Y \Rightarrow X \leq_{ts} Y \wedge X <_{pt} Y \Rightarrow X <_m Y$. So Rule 1 in Definition 1 is obeyed.

$S(a) \leq_{ts} L(a) \Rightarrow S(a) <_{ts} L(a) \vee S(a) =_{ts} L(a)$. By Invariant 3 in Theorem 1, this implies $S(a) \leq_{ts} L(a) \Rightarrow S(a) <_{ts} L(a) \vee S(a) =_{ts} L(a) \wedge S(a) <_{pt} L(a)$. Thus, from Definition 3, $S(a) \leq_{ts} L(a) \Rightarrow S(a) <_m L(a)$. We also have $S(a) <_m L(a) \Rightarrow S(a) \leq_{ts} L(a)$ from Definition 3. So $\{S(a)|S(a) \leq_{ts} L(a)\} = \{S(a)|S(a) <_m L(a)\}$. According to Invariant 2 in Theorem 1, all the elements in $\{S(a)|S(a) <_m L(a)\}$ have different timestamps, which means $<_m$ and $<_{ts}$ indicate the same ordering. Finally, Invariant 1 in Theorem 1 becomes Rule 2 in Definition 1. □

In the following two sections, we focus on the proof of Theorem 1.

3.2 Basic Lemma

We first give the definition of a clean block for ease of discussion.

Definition 4 (Clean Block)**.** *A clean block can be an L2 cacheline in S state, or an L1 cacheline in M state, or a ToM or WBRp message in a network buffer.*

Lemma 1. *∀ address a, at most one clean block exists.*

The basic lemma is an invariant about the cacheline states and the messages in network buffers. No timestamps are involved.

A visualization of Lemma 1 is shown in Fig. 2 where a solid line represents a clean block. When the L2 state for an address is S, no L1 can have that address in M state, and no *ToM* and *WBRp* may exist. Otherwise if the L2 state is M, either a *ToM* response exists, or an L1 has the address in M state, or a *WBRp* exists. Intuitively, Lemma 1 says only one block in the system can represent the latest data value.

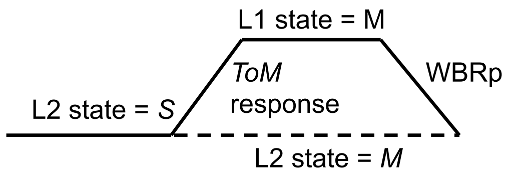

Figure 2: A visualization of Lemma 1

Lemma 1 Proof. For the base case, the lemma is trivially true since exactly one clean block exists for each address and the block is an L2 cacheline in S state. We now consider all the possible transition rules that may create a new clean block.

Only the *ExReq_S* rule can create a *ToM* response. However, the rule changes the state of the L2 cacheline from S to M and thus removes a clean block. Only the *L2Resp* rule can change an L1 cacheline state to M. However, it removes a *ToM* response from the *p2c* buffer. Both *Downgrade* and *WriteBackReq* can enqueue *WBRp* messages and both will change the L1 cacheline state from M to S or I. Only *WriteBackResp* changes the L2 cacheline to S state but it also dequeues a *WBRp* from the buffer.

In all of these transitions, a clean block is created and another one is removed. By the induction hypothesis, at most one clean block per address exists before the current transition, and still at most one clean block for the address exists after the transition. For other transitions not listed above, no clean block can be created so at most one clean block per address exists after any transition, proving the lemma. □

3.3 Timestamp Lemmas

Lemma 2. *At the current physical time, a clean block has the following invariants.*

Invariant 1 *Its rts is no less than the rts of all the other blocks (in caches and messages) with respect to the same address.*

Invariant 2 *Till the current physical time, no store has happened to the address at timestamp ts such that ts > rts.*

Proof. We prove the lemma by induction on the transition sequence. For the base case, only one block exists per address so Invariant 1 is true. All the stores so far happened at timestamp 0 which equals the *rts* of all the clean blocks, proving Invariant 2.

According to Lemma 1, for an address, exactly one clean block exists. By the induction hypothesis, if no timestamp changes and no clean block is generated, Invariant 1 is still true after the transition. By the transition rules, *wts* or *rts* can only be increased if the block is an L2 cacheline in the S state or an L1 cacheline in the M state. In both cases the block is clean. After the transition, the *rts* of the clean block increases and is still no less than the *rts* of other cachelines with the same address.

Similarly, by the induction hypothesis, Invariant 2 is true after the transition if no store happens and no clean block is generated. Only *StoreHit* can perform a store to a clean block, which changes both *wts* and *rts* to be $\max(pts, rts + 1)$. For the stored cacheline, since no store has happened with timestamp ts such that $ts > old\_rts$ (induction hypothesis), after the transition, no store, including the current one, has happened with timestamp ts such that $ts > \max(pts, old\_rts + 1) > old\_rts$.

Consider the last case where a clean block is generated at the current transition. Here, according to Lemma 1, another clean block disappears. In all the transitions, the *rts* of the new clean block equals the *rts* of the old block. Thus, both invariants are still true after the transition. □

Lemma 3. *For any block B in a cache or a message (WBRp, ToS and ToM), the data value associated with the block comes from a store St which has happened before the current physical time, and no other store St′ has happened such that $St.ts < St'.ts \le B.rts$, where St.ts is the timestamp of the store St and B.rts is the rts of block B.*

Proof. We prove the lemma by induction on the transition sequence. For the base case, each block has a corresponding store which happened before the system started and is thus before the current physical time. It is also the only store happened per address. Therefore the hypothesis is true.

We first prove that after a transition, for each block, there exists a store St which creates the data of the block and St happened before the current physical time. Consider the case where the data of a block does not change or comes from another block. Then, by the induction hypothesis, St must exist for the block after the transition. The only transition that changes the data of a block is *StoreHit*. After the store, however, the data stored in the cacheline comes from the current store which has just happened in physical time.

We now prove the second part of the lemma, that for any block B, no store $St' \neq St$ has happened such that $St.ts < St'.ts \le B.rts$. By the induction hypothesis, for the current transition, if no *data* or *rts* is changed in any block or if a block copies *data* and *rts* from an existing block, then the hypothesis is still true after the transition. The only cases in which the hypothesis may be violated are when the current transition changes *rts* or *data* for some block, which is only possible for *LoadHit*, *StoreHit* and *ShReq_S*.

For *LoadHit*, if the cacheline is in the S state, then *rts* remains unchanged. Otherwise, the cacheline must be a clean block, in which case *rts* is increased. Similarly, *ShReq_S* increases the *rts* and the cacheline must be a clean block again. By Invariant 2 in Lemma 2, no store has happened to the address with timestamp greater than *rts*. And thus after the *rts* is increased, no store can have happened with timestamp between the old *rts* to the new *rts*. By the induction hypothesis, we also have that no store St' could have happened such that $St.ts < St'.ts \le old\_rts$. These two inequalities together prove the hypothesis.

For *StoreHit*, both *rts* and *data* are modified. For the stored cacheline, after the transition, $St.ts = wts = rts = max(pts,\ old\_rts + 1)$. Thus, no St' may exist in this situation. For all the other cachelines, by Invariant 1 in Lemma 2, their *rts* is no greater than the old *rts* of the stored cacheline and is thus smaller than the timestamp of the current store. By the induction hypothesis, no store St' exists for those blocks before the transition. Thus, in the overall system, no such store St' can exist, proving the lemma. □

Finally, we prove Theorem 1.

Theorem 1 Proof. According to Lemma 3, for each $L(a)$, the loaded data is provided by an $S(a)$ and no other store $S'(a)$ has happened between the timestamp of $S(a)$ and the *rts*. And thus no $S'(a)$ has happened between the timestamp of $S(a)$ and the timestamp of the load which is no greater than *rts* by the transition rules. Therefore, Invariant 1 in Theorem 1 is true.

By the transition rules, a new store can only happen to a clean block and the timestamp of the store is $\max(pts, rts + 1)$. By Invariant 2 in Lemma 2, for a clean block at the current physical time, no store to the same address has happened with timestamp greater than the old rts of the cacheline. Therefore, for each new store, no store to the same address so far has the same timestamp as the new store, because the new store's timestamp is strictly greater than the old rts. And thus no two stores to the same address may have the same timestamp, proving Invariant 2.

Finally, we prove Invariant 3. If $S(a) =_{ts} L(a)$, by Invariant 1 in Theorem 1, $L(a)$ returns the data stored by $S(a)$. Then by Lemma 3, the store $S(a)$ must have happened before $L(a)$ in the physical time. □

4 Deadlock and Livelock Freedom

In this section, we prove that the Tardis protocol specified in Section 2 is both deadlock-free (Section 4.1) and livelock-free (Section 4.2).

4.1 Deadlock Freedom

Theorem 3 (Deadlock Freedom)**.** *After any sequence of transitions, if there is a pending request from any processor, then at least one transition rule (other than the Downgrade rule) can fire.*

Before proving the theorem, we first introduce and prove several lemmas.

Lemma 4. *If an L1 cacheline is busy, either a GetS or GetM request exists in its c2pRq buffer or a ToS or ToM response exists in its p2c buffer.*

Proof. This can be proven through induction on the transition sequence. In the base case, all the L1 cachelines are non-busy and the hypothesis is true. An L1 cacheline can only become busy through the *L1Miss* rule, which enqueues a request to its *c2pRq* buffer. A request can only be dequeued from *c2pRq* through the *ShReq_S* or *ExReq_S* rule, which enqueues a response into the same L1's *p2c* buffer. Finally, whenever a message is dequeued from the *p2c* buffer (*L2Resp* rule), the L1 cacheline becomes non-busy, proving the lemma. □

Lemma 5. *If an L2 cacheline is busy, the cacheline must be in state M.*

Proof. This lemma can be proven by induction on the transition sequence. For the base case, no cachelines are busy and the hypothesis is true. Only *Req_M* makes an L2 cacheline busy but the cacheline must be in the *M* state. Only *WriteBackResp* downgrades an L2 cacheline from the *M* state but it also makes the cacheline non-busy. □

Lemma 6. *For an L2 cacheline in the M state, the id of the clean block for the address equals the owner of the L2 cacheline.*

Proof. According to Lemma 1, exactly one clean block exists for the address. If the L2 state is *M*, the clean block can be a *ToM* response, an L1 cacheline in the *M* state or a *WBRp*. We prove the lemma by induction on the transition sequence.

The base case is true since no L2 cachelines are in the *M* state. We only need to consider cases wherein a clean block is created. When *ToM* is created (*ExReq_S* rule), its *id* equals the owner in the L2 cacheline. When an L1 cacheline in the *M* state is created (*L2Resp* rule), its *id* equals the *id* of the *ToM* response. When a *WBRp* is created (*WriteBackReq* or *Downgrade* rule), its *id* equals the *id* of the L1 cacheline. By the induction hypothesis, the *id* of a newly created clean block always equals the *owner* in the L2 cacheline which does not change as long as the L2 cacheline is in the *M* state. □

Lemma 7. *For a busy cacheline in L2, either a WBRq or a WBRp exists for the address with id matching the owner of the L2 cacheline.*

Proof. We prove the lemma by induction on the transition sequence. For the base case, no cacheline is busy and thus the hypothesis is true. We only need to consider the cases where an L2 cacheline is busy after the current transition, i.e., $\neg busy \Rightarrow busy$ and $busy \Rightarrow busy$.

Only the *Req_M* rule can cause a $\neg busy \Rightarrow busy$ transition and the rule enqueues a *WBRq* into *p2c* with *id* matching the *owner* and therefore the hypothesis is true.

For $busy \Rightarrow busy$, the lemma can only be violated if a *WBRq* or *WBRp* with matching *id* is dequeued. However, when a *WBRp* is dequeued, the cacheline becomes non-busy in L2 (*WriteBackResp* rule). If a *WBRq* is dequeued and the L1 cacheline is in the *M* state, a *WBRp* is created with a matching *id*. So the only case to consider is when the *WBRq* with matching *id* is dequeued, and the L1 cacheline is in the *S* or *I* states, and no other *WBRq* exists in the same *p2c* buffer and no *WBRp* exists in the *c2pRp* buffer.

The L2 cacheline can only become *busy* by sending a *WBRq*. The fact that the dequeued *WBRq* is the only *WBRq* in the *c2pRq* means that the L2 cacheline has been busy since the dequeued *WBRq* was sent (otherwise another *WBRq* will be sent when the L2 cacheline becomes busy again). Since *p2c* is a FIFO, when the *WBRq* is dequeued, the messages in the *p2c* must be sent after the *WBRq* was sent. By transition rules, the L2 cacheline cannot send *ToM* while being busy, so no *ToM* may exist in the *p2c* buffer when *WBRq* dequeues. As a result, no clean block exists with $id = owner$. Then, by Lemma 6, no clean block exists for the address (L2 is in the *M* state because of Lemma 5) which contradicts Lemma 1. □

Theorem 3 Proof. If any message exists in the *c2pRp* buffer, the *WriteBackResp* rule can fire. Consider the case where no message exists in *c2pRp* buffer. If any message exists in the *p2c* buffer's head, the *L2Resp* rule can fire, or the *WriteBackReq*, *LoadHit* or *StoreHit* rule can fire. For the theorem to be violated, no messages can exist in the *c2pRp* or *p2c* buffer. Then, according to Lemma 7, all cachelines in L2 are non-busy.

Now consider the case when no message exists in *c2pRp* buffer or *p2c* buffer and a *GetS* or *GetM* request exists in *c2pRq* for an L1 cache. Since the L2 is not busy, one of *ShReq_S*, *ExReq_S* and *Req_M* can fire, which enqueues a message into the *p2c* buffer.

Consider the last case where there is no message in any network buffer. By Lemma 4, all L1 cachelines are non-busy. By the hypothesis, there must be a request in *mRq* for some processor. Now if the request is a hit, the corresponding hit rule (*LoadHit* or *StoreHit*) can fire. Otherwise, the *L1Miss* rule can fire, sending a message to *c2pRq*. □

4.2 Livelock Freedom

Even though the Tardis protocol correctly follows sequential consistency and is deadlock-free, livelock may still occur if the protocol is not well designed. For example, for an L1 miss, the *Downgrade* rule may fire immediately after the *L2Resp* but before any *LoadHit* or *StoreHit* rule fires. As a result, the *L1Miss* needs to be fired again but the *Downgrade* always happens after the response comes back, leading to livelock. We avoid this possibility by only allowing *Downgrade* to fire when neither *LoadHit* nor *StoreHit* can fire.

To rigorously prove livelock freedom, we need to guarantee that some transition rule should eventually make forward progress and no transition rule can make backward progress. We give the following definition of livelock freedom.

Theorem 4. *After any sequence of transitions, if there exists a pending request from any processor, then within a finite number of transitions, some request at some processor will dequeue.*

In order to prove the theorem, we will show that for every transition rule, at least one request will make forward progress and move one step towards the end of the request and at the same time no other request makes backward progress; or if no request makes forward or backward progress for the transition, we show that such transitions can only be fired a finite number of times. Specifically, we define forward progress as a lattice of system states where each request in *mRq* (load or store) has its own lattice. Table 4 shows the lattice for a request where the lower parts in the lattice correspond to the states with more forward progress. We will prove livelock freedom by showing that for any state transition in any cache, any request either moves down the lattice (making forward progress) or stays at the current position but never moves up. Moreover, transitions which keep the state of every request staying at the same position in the lattice can only occur a finite number of times. Specifically, we will prove the following lemma.

Table 4: Lattice for a request. For a load request, $L1.miss \triangleq (L1.state = I \lor L1.state = S \land pts > L1.rts)$. For a store request, $L1.miss \triangleq (L1.state < M)$. *bufferName_exist* means a message exists in the buffer and *bufferName_rdy* means that the message is the head of the buffer. *bufferName_rdy* implies *bufferName_exist*.

1	L1.miss $\land$ ¬L1.busy
2	L1.miss $\land$ L1.busy $\land$ c2pRq_exist $\land$ ¬ c2pRq_rdy
3	L1.miss $\land$ L1.busy $\land$ c2pRq_rdy $\land$ L2.state = M $\land$ ¬L2.busy
4	L1.miss $\land$ L1.busy $\land$ c2pRq_rdy $\land$ L2.state = M $\land$ L2.busy $\land$ p2cRq_exist $\land$ ¬p2cRq_rdy
5	L1.miss $\land$ L1.busy $\land$ c2pRq_rdy $\land$ L2.state = M $\land$ L2.busy $\land$ p2cRq_rdy $\land$ ownerL1.state = M
6	L1.miss $\land$ L1.busy $\land$ c2pRq_rdy $\land$ L2.state = M $\land$ L2.busy $\land$ p2cRq_rdy $\land$ ownerL1.state < M
7	L1.miss $\land$ L1.busy $\land$ c2pRq_rdy $\land$ L2.state = M $\land$ L2.busy $\land$ ¬p2cRq_exist
8	L1.miss $\land$ L1.busy $\land$ c2pRq_rdy $\land$ L2.state = S
9	L1.miss $\land$ L1.busy $\land$ p2cRp_exist $\land$ ¬p2cRp_rdy
10	L1.miss $\land$ L1.busy $\land$ p2cRp_rdy
11	¬L1.miss

Lemma 8. *For a state transition except Downgrade, WriteBackReq and WriteBackResp, either a request dequeues from the mRq or at least one request will move down its lattice. For all the state transitions, no request will move up its lattice. Further, the system can only fire Downgrade, WriteBackReq and WriteBackResp for a finite number of times without firing other transitions.*

We need the following lemmas before proving Lemma 8.

Lemma 9. *If an L1 cacheline is busy, then exactly one request (GetS or GetM in c2pRq) or response (ToS or ToM in p2c) exists for the address and the L1 cache. If the L1 cacheline is non-busy, then no request or response can exist in its c2pRq and p2c.*

Proof. This lemma is a stronger lemma than Lemma 4. We prove this by the induction on the transition sequence. For the base case, all the L1 cachelines are non-busy and no message exists and thus the lemma is true.

We only need to consider the cases where the busy flag changes or any request or response is enqueued or dequeued. Only the *L1Miss*, *L2Resp*, *ShReq_S* and *ExReq_S* rules need to be considered.

For *L1Miss*, a request is enqueued to *c2pRq* and the L1 cacheline becomes busy. For *L2Resp*, a response is dequeued and the L1 cacheline becomes non-busy. For *ShReq_S* and *ExReq_S*, a request is dequeued but a response in enqueued. By the induction hypothesis, after the current transition, the hypothesis is still true for all the cases above, proving the lemma. □

Lemma 10. *If an L1 cacheline is busy, there must exist a request at the head of the mRq buffer for the address and the request misses in the L1.*

Proof. For the base case, all L1 cachelines are non-busy and the lemma is true.

We consider cases where the L1 cacheline is busy after the transition. Only *L1Miss* can make an L1 cacheline busy from non-busy and the rule requires a request to be waiting at the head of the *mRq* buffer. If the L1 cacheline stays busy, then no rule can remove the request from the *mRq* buffer. By the induction hypothesis, the lemma is true after any transition. □

Lemma 11. *If an L2 cacheline is busy, there must exist a request with the same address at the head of the c2pRq buffer in L2.*

Proof. The proof follows the same structure as the previous proof for Lemma 10. □

Lemma 12. *For a memory request in a c2pRq buffer, its type and pts equal the type and pts of a pending processor request to the same address at the head of the mRq at the L1 cache.*

Proof. By Lemmas 9 and 10, the L1 cacheline with the same address must be *busy* and a pending processor request exists at the head of the *mRq* buffer. Only the *L1Miss* rule sets the *type* and *pts* of a memory request in a *c2pRq* buffer and they equal the *type* and *pts* from the processor request. □

Lemma 13. *For a memory response in a p2c buffer, its type equals the type of a pending processor request to the same address at the L1 cache and if the type = S, its rts is no less than the pts of that processor request.*

Proof. Similar to the proof of Lemma 12, the processor request must exist. Only the *ShReq_S* and *ExReq_S* rules set the *type* and *rts* of the response, and *type* equals the *type* of a memory request and if $type = S$, *rts* is no less than the memory request. Then the lemma is true by Lemma 12. □

Lemma 14. *When the L2Resp rule fires, a request with the same address at the head of mRq will transition from an L1 miss to an L1 hit.*

Proof. Before the transition of *L2Resp*, the L1 cacheline is busy, and a response is at the head of the *p2c* buffer. By Lemma 13, if the pending processor request has $type = M$, then the memory response also has $type = M$ and thus it is an L1 hit. If the pending processor has $type = S$, also by Lemma 13, the memory response has $type = S$ and the *rts* of the response is no less than the *pts* of the pending request. Therefore, it is also an L1 hit. □

Lemma 15 (Coverage)**.** *The union of all the entries in Table 4 is True.*

Proof. By Lemma 4, if *L1.busy* we can prove that $c2pRq\_exist \vee p2cRp\_exist \Rightarrow True$.

Then it becomes obvious that the union of all the entries is true. □

Lemma 16 (Mutually Exclusive)**.** *The intersection of any two entries in Table 4 is False.*

Proof. For most pairs of entries, we can trivially check that the intersection is *False*. The only tricky cases are the intersection of entry 9 or 10 with an entry from 3 to 8. These cases can be proven *False* using Lemma 9, which implies that $c2pRq\_exist \wedge p2cRp\_exist \Rightarrow False$. □

Lemma 8 Proof. We need to prove two goals. First, for each transition rule except *Downgrade*, *WriteBackReq* and *WriteBackResp*, at least one request will dequeue or move down the lattice. Second, for all transition rules no request will move up the lattice.

We first prove that a transition with respect to address a_1 never moves a request with address a_2 ($\neq a_1$) up its lattice. The only possible way that the transition affects the request with a_2 is by dequeuing from a buffer which may make a request with a_2 being the head of the buffer and thus becomes ready. However, this can only move the request with a_2 down the lattice.

Also note that each processor can only serve one request per address at a time, because the *mRq* is a FIFO. Therefore, for the second goal we only need to prove that requests with the same address in other processors do not move up the lattice. We prove both goals for each transition rule.

For *LoadHit* and *StoreHit*, a request always dequeues from the *mRq* and the lemma is satisfied.

For the *L1Miss* rule, a request must exist and be in entry 1 in a table before the transition. And since $busy = True$ after the transition, it must move down the lattice to one of entries from 2 to 10. Since the L1 cacheline state does not change, no other requests in other processors move in their lattice.

For the *L2Resp* rule, according to Lemma 14, a request will move from *L1.miss* to *L1.hit*. In the lattice, this corresponds to moving from entry 10 to entry 11, which is a forward movement. For another request to the same address, the only entries that might be affected are entry 4, 5 and 6. However, since *p2c* is a FIFO and the response is ready in the *p2c* buffer before the transition, no *WBRq* can be ready in this *p2c* buffer for other requests with the same address and thus they cannot be in entry 5 or 6. If another request is in entry 4, the transition removes the response from the *p2c* and this may make the *WBRq* ready in *p2c* and thus the request moves down the lattice. In all cases, no other requests move up the lattice.

For the *ShReq_S* or *ExReq_S* rule to fire, there exists a request in the *c2pRq* buffer which means the address must be busy in the corresponding L1 (Lemma 9) and thus a request exists in its *mRq* and misses the L1 (Lemma 10). This request, therefore, must be in entry 8 in Table 4. The transition will dequeue the request and enqueue a response to *p2c* and thus moves the request down to entry 9 or 10. For all the other requests with the same address, they cannot be ready in the *c2pRq* buffer since the current request blocks them, and thus they are not in entry 3 to 8 in the lattice. For the other entries, they can only possibly be affected by the transition if the current request is dequeued and one of them becomes ready. This, however, only moves the request down the lattice.

Table 5: System Components required for main memory.

Component	Format	Message Types
MemRq	$MemRq.entry = (type,\ addr,\ data)$	*MemRq*
MemRp	$MemRp.entry = (addr,\ data)$	*MemRp*
Mem	$Mem[addr] = (data)$	-
mts	-	-

The *Req_M* rule can only fire if a request is ready in *c2pRq* and the L2 is in the *M* state. According to Lemma 9 and Lemma 10, there exists a request in one *mRq* that is in entry 3 in a table. After the transition, this request will move to entry 4 or 5 or 6 and thus down the lattice. For all the other requests, similar to the discussion of *ShReq_S* and *ExReq_S*, they either stay in the same entry or move down the lattice.

Finally, we talk about the *Downgrade*, *WriteBackReq* and *WriteBackResp* rules. The *Downgrade* rule can only fire when the L1 cacheline is non-busy, corresponding to entry 1 and 11 if the request is from the same L1 as the cacheline being downgraded. Entry 1 cannot move up since it is the first entry. If a request is in entry 11, since it is an L1 hit now, the *Downgrade* rule does not fire. For a request from a different L1, the *Downgrade* rule may affect entry 5 and 6. However, it can only move the request from entry 5 to 6 rather than the opposite direction.

For the *WriteBackReq* rule, if the L1 cacheline is in the *S* state, then nothing changes but a message is dequeued from the *p2c* buffer which can only move other requests down the lattice. If the L1 cacheline has *M* state, then if a request to the same address exists in the current L1, the request must be a hit and thus *WirteBackReq* cannot fire. For requests from other L1s, they can only be affected if they are in entry 4. Then, the current transition can only move them down the lattice.

For the *WriteBackResp* rule, the L2 cacheline moves from the *M* to the *S* state. All the other requests can only move down their lattice due to this transition.

Finally, we prove that *Downgrade*, *WriteBackReq* and *WriteBackResp* can only fire a finite number of times without other transitions being fired. Each time *Downgrade* is fired, an L1 cacheline's state goes down. Since there are only a finite number of L1 cachelines and a finite number of states, *Downgrade* can only be fired a finite number of times. Similarly, each *WriteBackReq* transition consumes a *WBRq* message which can only be replenished by the *Req_M* rule. And each *WriteBackResp* transition consumes a *WBRp* which is replenished by *Downgrade* and *WriteBackReq* and thus only has finite count. □

Theorem 4 Proof. If there exists a pending request from any processor, by Lemma 8, some pending request will eventually dequeue or move down the lattice which only has a finite number of states. For a finite number of processors, since the *mRq* is a FIFO, only a finite number of pending requests can exist. Therefore, some pending request will eventually reach the end of the lattice and dequeue, proving the theorem. □

5 Main Memory

For ease of discussion, we have assumed that all the data fit in the L2 cache, which is not realistic for some shared memory systems. A multicore processor, for example, has an offchip main memory which does not contain timestamps. For these systems, the components in Table 5 and transition rules in Table 6 need to be added. And for the initial system state, all the data should be in main memory with all L2 cachelines in *I* state, and $mts = 0$.

Most of the extra components and rules are handling main memory requests and responses and the *I* state in the L2. However, *mts* is a special timestamp added to represent the largest *rts* of the cachelines stored in the main memory. The *mts* guarantees that cachelines loaded from the main memory have proper timestamps and thus can be properly ordered.

Due to limited space, we only prove that the Tardis protocol with main memory still obeys sequential consistency (SC). The system can also be shown to be deadlock- and livelock-free using proofs similar to Section 4. For the SC proof, we only need to show that Lemma 1, 2 and 3 are true after the main memory is added.

In order to prove these lemmas, we need the following simple lemma.

Table 6: State Transition Rules for Main Memory.

Rules and Condition	Action
L2Miss **let** $(id,\ type,\ addr,\ pts) = c2pRq.get\_msg()$ **condition:** $L2.[addr].state = I$	MemRq.enq(S, $addr$, _) $busy := True$
MemResp **let** $(addr,\ data) =$ MemRp **let** $(state,\ l2data\ busy,\ owner,\ wts,\ rts) = L2.[addr]$	$state := S$ $l2data := data$ $busy := False$ $wts := mts$ $rts := mts$
L2Downgrade **let** $(state,\ data,\ busy,\ owner,\ wts,\ rts) = L2.[addr]$ **condition:** $state = M \wedge busy = False$	p2c.enq($owner$, Req, $addr$, _, _, _, _) $busy := True$
L2Evict **let** $(state,\ data,\ busy,\ owner,\ wts,\ rts) = L2.[addr]$ **condition:** $state = S$	$MemRq$.enq(M, $addr$, $data$) $state := I$ $mts := \max(rts,\ mts)$

Lemma 17. *If an L2 cacheline is in the I state, no clean block exists for the address.*

Proof. We prove by induction on the transition sequence. The hypothesis is true for the base case since no clean block exists. If an L2 cacheline moves from S to I (through the L2Evict rule), the clean block (L2 cacheline in S state) is removed and no clean block exists for that address. By the transition rules, while the L2 line stays in the I state, no clean block can be created. By the induction hypothesis, if an L2 cacheline is in the I state after a transition, then no clean block can exist for that address. □

For Lemma 1, we only need to include Lemma 17 in the original proof. For Lemmas 2 and 3, we need to show the following properties of mts.

Lemma 18. *If an L2 cacheline is in the I state, then the following statements are true.*

- *mts is no less than the rts of all the copies of the block.*
- *No store has happened to the address at ts such that ts > mts.*
- *The data value of the cacheline in main memory comes from a store St which happened before the current physical time. And no other store St′ has happened such that $St.ts < St'.ts \leq mts$.*

Proof. All the statements can be easily proven by induction on the transition sequence. For $S \to I$ of an L2 cacheline, since the end mts is no less than the rts, by Lemmas 2 and 3, all three statements are true after the transition.

Consider the other case where the L2 cacheline stays in I state. Since no clean block exists (Lemma 17), the copies of the cacheline cannot change their timestamps and no store can happen. By the transition rules, the mts never decreases after the transition. So the hypothesis must be true after the transition. □

To finish the original proof, we need to consider the final case where the L2 cacheline moves from I to S state (*MemResp* rule). In this case, both wts and rts are set to mts. By Lemma 18, both Lemma 2 and Lemma 3 are true after the current transition.

6 Related Work

Snoopy [15] and directory [7,35] cache coherence are both popular coherence protocols and are widely adopted in multicore processors [1,10], multi-socket systems [3,40] and distributed shared memory systems [19,24]. The Tardis protocol [37] is a different yet as powerful coherence protocol. Tardis is conceptually simpler than a directory protocol and has excellent scalability. Some other timestamp based coherence protocols have also been proposed in the literature [13,25,30,33] but none of them are as simple and high performant as Tardis.

Both model checking and formal proofs are popular in proving the correctness of cache coherence protocols. Model checking based verification [4,5,8,9,11,12,14,16–18,23,27–29,38,39] is a commonly used technique, but even with several optimizations, it does not scale to automatically verify real-world systems.

Many other works [2,6,21,26,32] prove the correctness of a cache coherence protocol by proving invariants as we did in this paper. Our invariants are in general simpler than what they had partly because Tardis is simpler than a directory coherence protocol. Finally, our proofs can be machine-checked along the lines of the proofs for a hierarchical cache coherence protocol [31, 36].

7 Conclusion

We provided simple, yet rigorous proofs of correctness for a recently-proposed scalable cache coherence protocol. Future work includes generalizing the protocol to relaxed memory consistency models and proving correctness, and machine-checking the proofs.

References

[1] Tile-gx family of multicore processors. `http://www.tilera.com`.

[2] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):182–205, 1993.

[3] D. Anderson and J. Trodden. *Hypertransport system architecture*. Addison-Wesley Professional, 2003.

[4] R. Bhattacharya, S. German, and G. Gopalakrishnan. Symbolic partial order reduction for rule based transition systems. In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 332–335. Springer Berlin Heidelberg, 2005.

[5] R. Bhattacharya, S. M. German, and G. Gopalakrishnan. Exploiting symmetry and transactions for partial order reduction of rule based specifications. In *In Antti Valmari, editor, SPIN, volume 3925 of Lecture Notes in Computer Science*, pages 252–270. Springer, 2006.

[6] G. M. Brown. Asynchronous multicaches. *Distributed Computing*, 4(1):31–36, 1990.

[7] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *Computers, IEEE Transactions on*, 100(12):1112–1118, 1978.

[8] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Form. Methods Syst. Des.*, 36(1):37–64, Feb. 2010.

[9] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *in Formal Methods in Computer Aided Design*, pages 382–398. Springer, 2004.

[10] G. Chrysos and S. P. Engineer. Intel xeon phi coprocessor (codename knights corner). In *Proceedings of the 24th Hot Chips Symposium, HC*, 2012.

[11] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer Berlin Heidelberg, 2000.

[12] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings, IEEE 1992 International Conference on*, pages 522–525, Oct 1992.

[13] M. Elver and V. Nagarajan. TSO-CC: Consistency directed cache coherence for TSO. In *International Symposium on High Performance Computer Architecture*, pages 165–176, 2014.

[14] E. A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, pages 247–262, 2003.

[15] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *ACM SIGARCH Computer Architecture News*, volume 11, pages 124–131. ACM, 1983.

[16] C.-W. N. Ip, D. L. Dill, and J. C. Mitchell. State reduction methods for automatic formal verification, 1996.

[17] R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, pages 396–410. Springer-Verlag, 2001.

[18] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. R. Tuttle, and Y. Yu. Checking cache-coherence protocols with TLA$^+$. *Formal Methods in System Design*, 22(2):125–131, 2003.

[19] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter*, volume 1994, 1994.

[20] G. Kurian, J. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. Kimerling, and A. Agarwal. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[21] E. Ladan-Mozes and C. E. Leiserson. A consistency architecture for hierarchical shared caches. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2008.

[22] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[23] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[24] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[25] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas. Memory coherence in the age of multicores. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 1–8. IEEE, 2011.

[26] M. M. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 182–193. IEEE, 2003.

[27] K. McMillan. Verification of infinite state systems by compositional model checking. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–237. Springer Berlin Heidelberg, 1999.

[28] K. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 111–134, 1992.

[29] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *In CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2144*, pages 179–195. Springer, 2001.

[30] S. Nandy and R. Narayan. An incessantly coherent cache scheme for shared memory multithreaded systems. Citeseer, 1994.

[31] S. Park and D. L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296. ACM Press, 1996.

[32] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport clocks: verifying a directory cache-coherence protocol. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 67–76. ACM, 1998.

[33] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache coherence for gpu architectures. pages 578–590, 2013.

[34] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[35] C. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 749–753. ACM, 1976.

[36] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave. Modular deductive verification of multiprocessor hardware designs. In *27th International Conference on Computer Aided Verification*, 2015. Accepted paper.

[37] X. Yu and S. Devadas. TARDIS: timestamp based coherence algorithm for distributed shared memory. *CoRR*, abs/1501.04504, 2015.

[38] M. Zhang, J. D. Bingham, J. Erickson, and D. J. Sorin. Pvcoherence: Designing flat coherence protocols for scalable verification. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 392–403. IEEE Computer Society, 2014.

[39] M. Zhang, A. R. Lebeck, and D. J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 471–482, Washington, DC, USA, 2010. IEEE Computer Society.

[40] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 1–6. IEEE, 2010.