Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory

Xiangyao Yu
CSAIL, MIT
Cambridge, MA, USA
yxy@mit.edu

Srinivas Devadas
CSAIL, MIT
Cambridge, MA, USA
devadas@mit.edu

***Abstract*—A new memory coherence protocol, Tardis, is proposed. Tardis uses timestamp counters representing logical time as well as physical time to order memory operations and enforce sequential consistency in any type of shared memory system. Tardis is unique in that as compared to the widely-adopted directory coherence protocol, and its variants, it completely avoids multicasting and only requires $O(\log N)$ storage per cache block for an N-core system rather than $O(N)$ sharer information. Tardis is simpler and easier to reason about, yet achieves similar performance to directory protocols on a wide range of benchmarks run on 16, 64 and 256 cores.**

***Index Terms*—coherence; timestamp; scalability; sequential consistency;**

I. Introduction

Shared memory systems are ubiquitous in parallel computing. Examples include multi-core and multi-socket processors, and distributed shared memory systems (DSM). The correctness of these systems is defined by the memory consistency model which specifies the legitimate interleaving of operations from different nodes (e.g., cores or processors). The enforcement of a consistency model heavily relies on the underlying coherence protocol. For a shared memory system, the coherence protocol is the key component to ensure performance and scalability.

When the data can be cached in the local memory of a node, most large-scale shared memory systems today adopt directory based coherence protocols [1], [2]. Examples include many-core systems with large core count [3], [4], coherence between multi-socket systems like Intel's QPI [5] and AMD's HyperTransport [6], and coherence of distributed shared memory systems like IVY [7] and Treadmarks [8]. A well known challenge in a directory coherence protocol is latency and scalability. For example, these protocols keep a list of nodes (sharers) caching each data and send invalidations to sharers before the data is modified by some node. Waiting for all invalidation requests to be acknowledged may take a long time and storing the sharer information or supporting broadcasting does not scale well as the number of nodes increases.

We propose a new coherence protocol, Tardis, which is simpler and more scalable than the simplest directory protocol, but has equivalent performance. Tardis *directly* expresses the memory consistency model by explicitly enforcing the global memory order using timestamp counters that represent logical as opposed to physical time; it does this without requiring a globally synchronized clock unlike prior timestamp coherence schemes (e.g., [9], [10]), and without requiring multicast/broadcast support unlike prior directory coherence schemes (e.g., [11], [12]). In Tardis, only the timestamps and the owner ID need to be stored for each address for a $O(\log N)$ cost where N is the number of processors or cores; the $O(N)$ sharer information of common directory protocols is not required. The requirement of storing sharers is avoided partly through the novel insight that a writer can instantly jump ahead[1] to a time when the sharer copies have expired and immediately perform the write without violating sequential consistency. A formal proof that Tardis satisfies sequential consistency can be found in [13].[2]

We evaluated Tardis in the context of multi-core processors. Our experiments showed that Tardis achieves similar performance to its directory counterpart over a wide range of benchmarks. Due to its simplicity and excellent performance, we believe Tardis is a competitive alternative to directory coherence for massive-core and DSM systems.

We provide background in Section II, describe the basic Tardis protocol in Section III, and optimizations to the basic protocol in Section IV. We evaluate Tardis in Section VI, discuss related work in Section VII and conclude the paper in Section VIII.

II. Background

In this section, we provide some background on memory consistency and coherence.

A. Sequential Consistency

A memory consistency model defines the correctness of a shared memory system. Specifically, it defines the legitimate behavior of memory loads and stores. Although a large number of consistency models exist, we will focus on *sequential consistency* due to its simplicity.

Sequential consistency was first proposed and formalized by Lamport [14]. A parallel program is sequentially consistent if *"the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core)*

[1] hence the name Tardis!

[2] The proof corresponds to a slightly simplified version of the protocol presented here.

appear in this sequence in the order specified by its program". If we use $<_p$ and $<_m$ to denote program order and global memory order respectively, sequential consistency requires the following two rules to be held [15]:

Rule 1: $X <_p Y \implies X <_m Y$

Rule 2:

$Value\ of\ L(a) = Value\ of\ Max_{<m}\{S(a)|S(a) <_m L(a)\}$

where $L(a)$ is a load to address a and $S(a)$ is a store to address a; the $Max_{<m}$ operator selects the most recent operation in the global memory order.

Rule 1 says that if an operation X (a load or a store) is before another operation Y in the program order of any core, X must precede Y in the global memory order. Rule 2 says that a load to an address should return the value of the most recent store to that address with respect to the global memory order.

B. Directory-Based Coherence

In practical systems, each core/processor has some private local storage to exploit locality. A memory coherence protocol is therefore used to support the consistency model.

Although both snoopy and directory protocols are used in small systems, virtually all large-scale shared memory systems today use some variant of the basic directory-based coherence protocol. The directory is a software or hardware structure tracking how the data are shared or owned by different cores/processors. In a directory protocol, the second rule of sequential consistency is achieved through the invalidation mechanism; when a core/processor writes to an address that is shared, all the shared copies need to be invalidated before the write can happen. Future reads to that address have to send requests to the directory which returns the value of the last write. This mechanism essentially guarantees that reads that happen after the last write with respect to physical time can only observe the value of the last write (the second rule of sequential consistency).

The directory needs to keep the sharer information of each address in order to correctly deliver the invalidations. If the system has N cores/processors, the canonical protocol requires $O(N)$ storage per address, which does not scale well when the system gets bigger. Alternative solutions to avoid $O(N)$ storage do exist (*cf.* Section VII) but either require broadcasting, incur significant additional complexity, or do not perform well.

III. Basic Protocol

We present a new coherence protocol, Tardis, which only requires $O(\log N)$ storage per cacheline and requires neither broadcasting/multicasting support nor a globally synchronized clock across the whole system. Tardis works for all types of distributed shared memory systems and is compatible with different memory consistency models.

In this paper, we present the Tardis protocol for sequential consistency in the context of a multi-core processor with shared last level cache (LLC). Our discussion applies equally well to other types of shared memory systems.

A. Timestamp Ordering

In a directory protocol (*cf.* Section II-B), the global memory order ($<_m$) is enforced through the physical time order. i.e., if X and Y are memory operations to the same address A and one of them is a store, then

$$X <_m Y \quad \implies \quad X <_{pt} Y$$

In Tardis, we break the correlation between the global memory order and the physical time order for *write after read* (WAR) dependencies while maintaining the correlation for *write after write* (WAW) and *read after write* (RAW) dependencies.

$$S_1(A) <_m S_2(A) \implies S_1(A) <_{pt} S_2(A)$$

$$S(A) <_m L(A) \implies S(A) <_{pt} L(A)$$

$$L(A) <_m S(A) \not\Longrightarrow L(A) <_{pt} S(A)$$

Tardis achieves this by explicitly assigning a timestamp to each memory operation to indicate its global memory order. Specifically, the global memory order in Tardis is defined as a combination of physical time and logical timestamp order, i.e., physi-logical time order, which we will call *physiological time order* for ease of pronunciation.

Definition 1 (Physiological Time Rule):

$$X <_m Y \quad := \quad X <_{ts} Y \ or\ (X =_{ts} Y\ and\ X <_{pt} Y)$$

In Definition 1 the global memory order is explicitly expressed using timestamps. Operations without dependency (e.g., two concurrent read operations) or with obvious relative ordering (e.g., accesses to private data from the same core) can share the same timestamp and their global memory order is implicitly expressed using the physical time order.

Using the physiological time rule, Rule 1 of sequential consistency becomes $X <_p Y \Rightarrow X <_{ts} \vee (X =_{ts} Y \wedge X <_{pt} Y)$. Assuming a processor always does in-order commit, we have $X <_p Y \Rightarrow X <_{pt} Y$. So Tardis only needs to guarantee that $X <_p Y \Rightarrow X \leq_{ts} Y$, i.e., operations from the same processor have monotonically increasing timestamps in the program order. For Rule 2 of sequential consistency, Tardis needs to guarantee that a load observes the correct store in the global memory order as defined by Definition 1. The correct store is the latest store – either the one with the largest logical timestamp or the latest physical time among the stores with the largest logical timestamp [13].

We note that the physiological timestamp here is different from Lamport clocks [16]. In Lamport clocks, a timestamp is incremented for each operation while a physiological timestamp is not incremented if the order is implicit in physical time. That said, the physiological timestamp does share some commonality with the Lamport clock. In a sense, Tardis applies Lamport/physiological timestamp to distributed shared memory systems.

TABLE I
TIMESTAMP MANAGEMENT IN THE TARDIS PROTOCOL WITHOUT PRIVATE MEMORY

Request Type	Load Request	Store Request
Timestamp Operation	$pts \Leftarrow Max(pts, wts)$ $rts \Leftarrow Max(pts, rts)$	$pts \Leftarrow Max(pts, rts + 1)$ $wts \Leftarrow pts$ $rts \Leftarrow pts$

B. Tardis without Private Cache

In Tardis, timestamps are maintained as logical counters. Each core keeps a program timestamp (*pts*) which is the timestamp of the last operation in the program order. Each cacheline keeps a read timestamp (*rts*) and a write timestamp (*wts*). The *rts* equals the largest timestamp among all the loads of the cacheline thus far and the *wts* equals the timestamp of the latest store to the cacheline. Tardis keeps the invariant that for a cacheline, its current data must be valid between its current *wts* and *rts*. The *pts* should not be confused with the processor clock, it does not increment every cycle and is not globally synchronized. The directory structure is replaced with a timestamp manager. Any load or store request to the LLC should go to the timestamp manager.

For illustrative purposes, we first show the Tardis protocol assuming no private cache and all data fitting in the shared LLC. Each cacheline has a unique copy in the LLC which serves all the memory requests. Although no coherence protocol is required in such a system, the protocol in this section provides necessary background in understanding the more general Tardis protocol in Section III-C.

Table I shows one possible timestamp management policy that obeys the two rules of sequential consistency. But other policies also exist. Each memory request contains the core's *pts* before the current memory operation. After the request, *pts* is updated to the timestamp of the current operation.

For a load request, the timestamp manager returns the value of the last store. According to Rule 1, the load timestamp must be no less than the current *pts*. According to Rule 2, the load timestamp must be no less than *wts* which is the timestamp of the last store to this cacheline. So the timestamp of the load equals $Max(pts, wts)$. If the final $pts > rts$, then *rts* bumps up to this *pts* since the *rts* should be the timestamp of the last read in the timestamp order.

For a store request, the last load of the cacheline (at *rts*) did not observe the value of the current store. According to Rule 2, the timestamp of the current store must be greater than the *rts* of the cacheline (the timestamp of the last load). So *pts* becomes $Max(pts, rts + 1)$. *wts* and *rts* should also bump up to this final *pts* since a new version has been created.

Both Rule 1 and Rule 2 hold throughout the protocol: the *pts* never decreases and a load always observes the correct store in the timestamp order.

C. Tardis with Private Cache

With private caching, data accessed from the LLC are stored in the private cache. The protocol introduced in Section III-B largely remains the same. However, two extra mechanisms need to be added.

Timestamp Reservation: Unlike the previous protocol where a load happens at a particular timestamp, timestamp reservation allows a load to reserve the cacheline in the private cache for a period of logical time (i.e., the lease). The end timestamp of the reservation is stored in *rts*. The cacheline can be read until the timestamp expires ($pts > rts$). If the cacheline being accessed has already expired, a request must be sent to the timestamp manager to extend the lease.

Exclusive Ownership: Like in a directory protocol, a modified cacheline can be exclusively cached in a private cache. In the timestamp manager, the cacheline is in exclusive state and the owner of the cacheline is also stored which requires $\log(N)$ bits of storage. The data can be accessed freely by the owner core as long as it is in the exclusive state; and the timestamps are properly updated with each access. If another core later accesses the same cacheline, a write back (the owner continues to cache the line in shared state) or flush request (the owner invalidates the line) is sent to the owner which replies with the latest data and timestamps.

Note that in the private cache, the meanings of *rts* for shared and exclusive cachelines are different. For a shared cacheline, *rts* is the end timestamp of the reservation; for an exclusive cacheline, *rts* is the timestamp of the last load or store. The state transition and the timestamp management of Tardis with private cache are shown in Table II and Table III. Table II shows the state transition at the private cache and Table III shows the state transition at the shared timestamp manager. Table IV shows the network message types used in the Tardis protocol where the suffix REQ and REP represent request and response respectively.

In the protocol, each cacheline (denoted as D) has a write timestamp (D.wts) and a read timestamp (D.rts). Initially, all *pts*'s and *mts*'s are 1 and all caches are empty. Some network messages (denoted as M or $reqM$) also have timestamps associated with them. Each message requires at most two timestamps.

We now discuss different cases of the Tardis protocol shown in both tables.

1) State Transition in Private Cache (Table II):

Load to Private Cache (column 1, 4, 5): A load to the private cache is considered as a hit if the cacheline is in exclusive state or is in shared state and has not expired ($pts \leq rts$). Otherwise, a SH_REQ is sent to the timestamp manager to load the data or to extend the existing lease. The request message has the current *wts* of the cacheline indicating the version of the cached data.

Store to Private Cache (column 2, 4, 5): A store to the private cache can only happen if the cacheline is exclusively owned by the core. Same as directory coherence, EX_REQ is sent to the timestamp manager for exclusive ownership. The *rts* and *wts* of the private data are updated to $Max(pts, rts+1)$ because the old version might be loaded at timestamp *rts* by another core.

Eviction (column 3): Evicting shared cachelines does not require sending any network message. The cacheline can simply be invalidated. Evicting exclusive cachelines is the

TABLE II

STATE TRANSITION IN PRIVATE CACHE. *TM* IS THE SHARED TIMESTAMP MANAGER, D IS THE DATA, M IS THE MESSAGE, $reqM$ IS THE REQUEST MESSAGE IF TWO MESSAGES ARE INVOLVED. TIMESTAMP TRANSITION IS HIGHLIGHTED IN **RED**.

States	Core Event			Network Event		
	Load	Store	Eviction	SH_REP or EX_REP	RENEW_REP or UPGRADE_REP	FLUSH_REQ or WB_REQ
Invalid	send SH_REQ to TM **M.wts⇐0,** **M.pts⇐pts**	send EX_REQ to TM **M.wts⇐0**		Fill in data **SH_REP** **D.wts⇐M.wts** **D.rts⇐M.rts** state⇐Shared **EX_REP** **D.wts⇐M.wts** **D.rts⇐M.rts** state⇐Excl.		
Shared $pts \leq rts$	Hit **pts⇐Max(pts, D.wts)**	send EX_REQ to TM **M.wts⇐D.wts**	state⇐Invalid No msg sent.		**RENEW_REP** **D.rts⇐M.rts** **UPGRADE_REP** **D.rts⇐M.rts** state⇐Excl.	
Shared $pts > rts$	send SH_REQ to TM **M.wts⇐D.wts,** **M.pts⇐pts**					
Exclusive	Hit **pts⇐Max(pts, D.wts)** **D.rts⇐Max(pts, D.rts)**	Hit **pts⇐Max(pts, D.rts+1)** **D.wts⇐pts** **D.rts⇐pts**	state⇐Invalid send FLUSH_REP to TM **M.wts⇐D.wts,** **M.rts⇐D.rts**			**FLUSH_REQ** **M.wts⇐D.wts** **M.rts⇐D.rts** send FLUSH_REP to TM state⇐Invalid **WB_REQ** **D.rts⇐Max(D.rts, D.wts+lease, reqM.rts)** **M.wts⇐D.wts** **M.rts⇐D.rts** send WB_REP to TM state⇐Shared

TABLE III

STATE TRANSITION IN TIMESTAMP MANAGER.

States	SH_REQ	EX_REQ	Eviction	DRAM_REP	FLUSH_REP or WB_REP
Invalid	Load from DRAM			Fill in data **D.wts⇐mts** **D.rts⇐mts** state⇐Shared	
Shared	**D.rts⇐Max(D.rts, D.wts+lease, reqM.pts+lease)** **if reqM.wts=D.wts** send RENEW_REP to requester **M.rts⇐D.rts** **else** send SH_REP to requester **M.wts⇐D.wts** **M.rts⇐D.rts**	**M.rts⇐D.rts** state⇐Excl. **if reqM.wts=D.wts** send UPGRADE_REP to requester **else** **M.wts⇐D.wts** send EX_REP to requester	**mts⇐Max(mts, D.rts)** Store data to DRAM if dirty state⇐Invalid		
Exclusive	send WB_REQ to the owner **M.rts⇐reqM.pts+lease**	send FLUSH_REQ to the owner			Fill in data **D.wts⇐M.wts,** **D.rts⇐M.rts** state⇐Shared

same as in directory coherence; the data is returned to the timestamp manager (through a FLUSH_REP message) and the cacheline is invalidated.

Flush or Write Back (column 6): Exclusive cachelines in the private cache may receive flush or write back requests from the timestamp manager if the cacheline is evicted from the LLC or accessed by other cores. A flush is handled similarly to an eviction where the data is returned and the line invalidated. For a write back request, the data is returned but the line becomes shared.

2) State Transition in Timestamp Manager (Table III):

Shared Request to Timestamp Manager (column 1): If the cacheline is invalid in LLC, it must be loaded from DRAM. If it is exclusively owned by another core, then a write back request is sent to the owner. When the cacheline is in the *Shared* state, it is reserved for a period of logical time by setting the *rts* to be the end timestamp of the reservation, and the line can only be read from *wts* to *rts* in the private cache.

If the *wts* of the request equals the *wts* of the cacheline in the timestamp manager, the data in the private cache must be the same as the data in the LLC. So a RENEW_REP is sent back to the requester without the data payload. Otherwise SH_REP is sent back with the data.

Exclusive Request to Timestamp Manager (column 2): An exclusive request can be either an exclusive load or exclusive store. Similar to a directory protocol, if the cacheline is invalid, it should be loaded from DRAM; if the line is exclusively owned by another core, a flush request should be sent to the owner.

If the requested cacheline is in shared state, however, *no invalidation messages need to be sent*. The timestamp manager can immediately give exclusive ownership to the requesting core which bumps up its local *pts* to be the current *rts* + 1 when it writes to the cacheline, i.e., jumps ahead in time. Other cores can still read their local copies of the cacheline if they have not expired. This does not violate sequential consistency since the read operations in the sharing cores are ordered before the write operation in physiological time though not necessarily

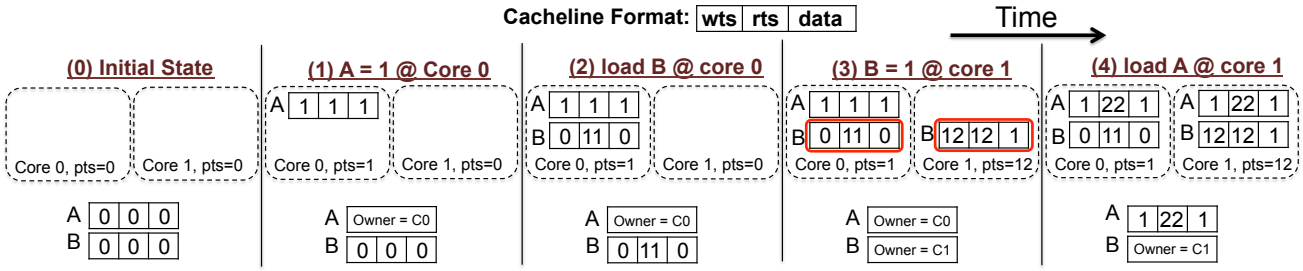


Fig. 1. An example program running with Tardis (*lease*= 10). Cachelines in private caches and LLC are shown. The cacheline format is at the top of the figure.

TABLE IV
NETWORK MESSAGES. THE CHECK MARKS INDICATE WHAT COMPONENTS THE MESSAGE CONTAINS.

Message Type	*pts*	*rts*	*wts*	data
SH_REQ	√		√	
EX_REQ			√	
FLUSH_REQ				
WB_REQ		√		
SH_REP		√	√	√
EX_REP		√	√	√
UPGRADE_REP		√		
RENEW_REP		√		
FLUSH_REP		√	√	√
WB_REP		√	√	√
DRAM_ST_REQ				√
DRAM_LD_REQ				
DRAM_LD_REP				√

in physical time. If the cacheline expires in the sharing cores, they will send requests to renew the line at which point they get the latest version of the data.

If the *wts* of the request equals the *wts* of the cacheline in the timestamp manager, the data is not returned and an UPGRADE_REP is sent to the requester.

Evictions (column 3): Evicting a cacheline in exclusive state is the same as in directory protocols, i.e., a flush request is sent to the owner before the line is invalidated. For shared cachelines, however, no invalidation messages are sent. Sharing cores can still read their local copies until they expire – this does not violate timestamp ordering.

DRAM (column 3, 4): Tardis only stores timestamps on chip but not in DRAM. The *memory timestamp* (*mts*) is used to maintain coherence for DRAM data. *mts* is stored per timestamp manager. It indicates the maximal read timestamp of all the cachelines mapped to this timestamp manager but evicted to DRAM. For each cacheline evicted from the LLC, *mts* is updated to be $Max(rts, mts)$. When a cacheline is loaded from DRAM, both its *wts* and *rts* are assigned to be *mts*. This guarantees that accesses to previously cached data with the same address are ordered before the accesses to the cacheline just loaded from DRAM. This takes care of the case when a cacheline is evicted from the LLC but is still cached in some core's private cache. Note that multiple *mts*'s can be stored per timestamp manager for different ranges of cacheline addresses. In this paper, we only consider a single *mts* per timestamp manager for simplicity.

Flush or write back response (column 5): Finally, the flush response and write back response are handled in the same way as in directory protocols. Note that when a cacheline is exclusively owned by a core, only the owner has the latest *rts* and *wts*; the *rts* and *wts* in the timestamp manager are invalid and the bits can be reused to store the ID of the owner core.

3) An Example Program: We use an example to show how Tardis works with a parallel program. Fig. 1 shows how the simple program in Listing 1 runs with the Tardis protocol. In the example, we assume a lease of 10 and that the instructions from Core 0 are executed before the instructions in Core 1.

Listing 1. Example Program

```
   initially A = B = 0
[Core 0]          [Core 1]
 A = 1             B = 1
 print B           print A
```

Step 1 : The store to A misses in Core 0's private cache and an EX_REQ is sent to the timestamp manager. The store operation should happen at $pts = Max(pts, A.rts + 1) = 1$ and the A.*rts* and A.*wts* in the private cache should also be 1. The timestamp manager marks A as exclusively owned by Core 0.

Step 2 : The load of B misses in Core 0's private cache. After Step 1, Core 0's *pts* becomes 1. So the reservation end timestamp should be $Max(rts, wts+lease, pts+lease) = 11$.

Step 3 : The store to B misses in Core 1's private cache. At the timestamp manager, the exclusive ownership of B is immediately given to Core 1 at *pts* = *rts* + 1 = 12. Note that two different versions of B exist in the private caches of core 0 and core 1 (marked in red circles). In core 0, $B = 0$ but is valid when $0 \leq$ *timestamp* ≤ 11; in Core 1, B = 1 and is only valid when *timestamp* ≥ 12. This does not violate sequential consistency since the loads of B at core 0 will be logically ordered before the loads of B at core 1, even if they may happen the other way around with respect to the physical time.

Step 4 : Finally the load of A misses in Core 1's private cache. The timestamp manager sends a WB_REQ to the owner (Core 0) which updates its own timestamps and writes back the data. Both cores will have the same data with the same range of valid timestamps.

With Tardis on sequential consistency, it is impossible for the example program above to output 0 for both A and B, even for out-of-order execution. The reason will be discussed in Section III-D.

D. Out-of-Order Execution

So far we have assumed in-order cores, i.e., a second instruction is executed only after the first instruction commits

and updates the *pts*. For out-of-order cores, a memory instruction can be executed before previous instructions finish and thus the current *pts* is not known. However, with sequential consistency, all instructions must commit in the program order. Tardis therefore enforces timestamp order at the commit time.

1) Timestamp Checking: In the re-order buffer (ROB) of an out-of-order core, instructions commit in order. We slightly change the meaning of *pts* to mean the timestamp of the last *committed* instruction. For sequential consistency, *pts* still increases monotonically. Before committing an instruction, the timestamps are checked. Specifically, the following cases may happen for shared and exclusive cachelines, respectively.

A shared cacheline can be accessed by load requests. And there are two possible cases.

1) *pts* $\leq$ *rts*. The instruction commits. *pts* $\Leftarrow Max(wts, pts)$.
2) *pts* $>$ *rts*. The instruction aborts and is restarted with the latest *pts*. Re-execution will trigger a renew request.

An exclusive cacheline can be accessed by both load and store requests. And the accessing instruction can always commit with *pts* $\Leftarrow Max(pts, wts)$ for a load operation and *pts* $\Leftarrow Max(pts, rts + 1)$ for a store operation.

There are two possible outcomes of a restarted load. If the cacheline is successfully renewed, the contents of the cacheline do not change. Otherwise, the load returns a different version of data and all the depending instructions in the ROB need to abort and be restarted. This hurts performance and wastes energy. However, the same flushing operation is also required for an OoO core on a baseline directory protocol under the same scenario [17]. If an invalidation happens to a cacheline after it is executed but before it commits, the load is also restarted and the ROB flushed. In this case, the renewal failure in Tardis serves as similar functionality to an invalidation in directory protocols.

2) Out-of-Order Example: If the example program in Section III-C3 runs on an out-of-order core, both loads may be scheduled before the corresponding stores making the program print $A = B = 0$. In this section, we show how this scenario can be detected by the timestamp checking at commit time and thus never happens. For the program to output $A = B = 0$ in Tardis, both loads are executed before the corresponding stores in the timestamp order.

$$L(A) <_{ts} S(A), \qquad L(B) <_{ts} S(B)$$

For the instruction sequence to pass the timestamp checking, we have

$$S(A) \leq_{ts} L(B), \qquad S(B) \leq_{ts} L(A)$$

Putting them together leads to the following contradiction.

$$L(A) <_{ts} S(A) \leq_{ts} L(B) <_{ts} S(B) \leq_{ts} L(A)$$

This means that at least at one core, the timestamp checking will fail. The load at that core is restarted with the updated *pts*. The restarted load will not renew the lease but return the latest value (i.e., 1). So at least at one core, the output value is 1 and $A = B = 0$ can never happen.

E. Avoiding Livelock

Although Tardis strictly follows sequential consistency, it may generate livelock due to deferred update propagation. In directory coherence, a write is quickly observed by all the cores through the invalidation mechanism. In Tardis, however, a core may still read the old cached data even if another core has updated it, as long as the cacheline has not expired. In other words, the update to the locally cached data is deferred. In the worst case when deferment becomes indefinite, livelock occurs. For example, if a core spins on a variable which is set by another core, the *pts* of the spinning core does not increase and thus the old data never expires. As a result, the core may spin forever without observing the updated data.

We propose a very simple solution to handle this livelock problem. To guarantee forward progress, we only need to make sure that an update is *eventually* observed by following loads, that is, the update becomes globally visible within some finite physical time. This is achieved by occasionally incrementing the *pts* in each core so that the old data in the private cache eventually expires and the latest update becomes visible. The self increment can be periodic or based on more intelligent heuristics. We restrict ourselves to periodic increments in this paper.

F. Tardis vs. Directory Coherence

In this section, we compare Tardis to the directory coherence protocol.

1) Protocol Messages: In Table II and Table III, the advantages and disadvantages of Tardis compared to directory protocols are shaded in light green and light red, respectively. Both schemes have similar behavior and performance in the other state transitions (the white cells).

Invalidation: In a directory protocol, when the directory receives an exclusive request to a *Shared* cacheline, the directory sends invalidations to all the cores sharing the cacheline and waits for acknowledgements. This usually incurs significant latency which may hurt performance. In Tardis, however, no invalidation happens (*cf.* Section III-C) and the exclusive ownership can be immediately returned without waiting. The timestamps guarantee that sequential consistency is maintained. The elimination of invalidations makes Tardis much simpler to implement and reason about.

Eviction: In a directory protocol, when a shared cacheline is evicted from the private cache, a message is sent to the directory where the sharer information is stored. Similarly, when a shared cacheline is evicted from the LLC, all the copies in the private caches should be invalidated. In Tardis, correctness does not require maintaining sharer information and thus no such invalidations are required. When a cacheline is evicted from the LLC, the copies in the private caches can still exist and be accessed.

Data Renewal: In directory coherence, a load hit only requires the cacheline to exist in the private cache. In Tardis, however, a cacheline in the private cache may have expired and cannot be accessed. In this case, a renew request is sent to the timestamp manager which incurs extra latency and network

traffic. In Section IV-A, we present techniques to reduce the overhead of data renewal.

2) Scalability: A key advantage of Tardis over directory coherence protocols is scalability. Tardis only requires the storage of timestamps for each cacheline and the owner ID for each LLC cacheline ($O(\log N)$, where N is the number of cores). In practice, the same hardware bits can be used for both timestamps and owner ID in the LLC; because when the owner ID needs to be stored, the cacheline is exclusively owned and the timestamp manager does not maintain the timestamps.

On the contrary, a canonical directory coherence protocol maintains the list of cores sharing a cacheline which requires $O(N)$ storage overhead. Previous works proposed techniques to improve the scalability of directory protocols by introducing broadcast or other complexity. They are discussed in Section VII-B.

3) Simplicity: Another advantage of Tardis is its conceptual simplicity and elegance. Tardis is directly derived from the definition of sequential consistency and the timestamps explicitly express the global memory order. This makes it easier to argue the correctness of the protocol. Concretely, given that Tardis does not have to multicast/broadcast invalidations and collect acknowledgements, the number of transient states in Tardis is smaller than that of a directory protocol.

IV. Optimizations and Extensions

We introduce several optimizations in the Tardis protocol in this section, which were enabled during our evaluation. The evaluation of the extensions described here is deferred to future work.

A. Speculative Execution

As discussed in Section III-F, the main disadvantage of Tardis compared to directory coherence protocol is the renew request. In a pathological case, the *pts* of a core may rapidly increase since some cachelines are frequently read-write shared by different cores. Meanwhile, the read-only cachelines will frequently expire and a large number of renew requests are generated incurring both latency and network traffic. Observe, however, that most renew requests will successfully extend the lease and the renew response does not transfer the data. This significantly reduces the network traffic of renewals. More important, this means that the data in the expired cacheline is actually valid and we could have used the value without stalling the pipeline of the core. Based on this observation, we propose the use of speculation to hide renew latency. When a core reads a cacheline which has expired in the private cache, instead of stalling and waiting for the renew response, the core reads the current value and continues executing speculatively. If the renewal fails and the latest cacheline is returned, the core rolls back by discarding the speculative computation that depends on the load. The rollback process is almost the same as a branch misprediction which has already been implemented in most processors.

For processors that can buffer multiple uncommitted instructions, successful renewals (which is the common case) do not hurt performance. Speculation failure does incur performance overhead since we have to rollback and rerun the instructions speculatively executed. However, if the same instruction sequence is executed in a directory protocol, the expired cacheline should not be in the private cache in the first place; the update from another core should have already invalidated this cacheline and a cache miss should happen. As a result, in both Tardis and directory coherence, the value of the load should be returned at the same time incurring the same latency and network traffic. Tardis still has some extra overhead as it needs to discard the speculated computation, but this overhead is relatively small.

Speculation successfully hides renew latency in most cases. The renew messages, however, may still increase the on-chip network traffic. This is especially problematic if the private caches have a large number of *shared* cachelines that all expire when the *pts* jumps ahead due to a write or self increment. This is a fundamental disadvantage of Tardis compared to directory coherence protocols. According to our evaluations in Section VI, however, Tardis has good performance and acceptable network overhead on real benchmarks even with this disadvantage. We leave solutions to pathologically bad cases to future work.

B. Timestamp Compression

In Tardis, all the timestamps increase monotonically and may roll over. One simple solution is to use 64-bit timestamps which never roll over in practice. This requires 128 extra bits to be stored per cacheline, which is a significant overhead. Observe, however, that the higher bits in a timestamp change infrequently and are usually the same across most of the timestamps. We exploit this observation and propose to compress this redundant information using a *base-delta* compression scheme.

In each cache, a *base timestamp* (*bts*) stores the common high bits of *wts* and *rts*. In each cacheline, only the *delta timestamps* (*delta_ts*) are stored (*delta_wts* = *wts* $-$ *bts* and *delta_rts* = *rts* $-$ *bts*). The actual timestamp is the sum of the *bts* and the corresponding *delta_ts*. The *bts* is 64 bits to prevent rollover; and there is only one *bts* per cache. The per cacheline *delta_ts* is much shorter to reduce the storage overhead.

When any *delta_ts* in the cache rolls over, we will rebase where the local *bts* is increased and all the *delta_ts* in the cache are decreased by the same amount, i.e., half of the maximum *delta_ts*. For simplicity, we assume that the cache does not serve any request during the rebase operation.

Note that increasing the *bts* in a cache may end up with some *delta_ts* being negative. In this case, we just set the *delta_ts* to 0. This effectively increases the *wts* and *rts* in the cacheline but it does not violate the consistency model. Consider a shared LLC cacheline or an exclusive private cacheline – the *wts* and *rts* can be safely increased. Increasing the *wts* corresponds to writing the same data to the line at a later logical time, and increasing the *rts* corresponds to a hypothetical read at a later logical time. Neither operation violates the rules of sequential consistency. Similarly, for

a shared cacheline in the private cache, *wts* can be safely increased as long as it is smaller than *rts*. However, *rts* can not be increased without coordinating with the timestamp manager. So if *delta_rts* goes negative in a shared line in a private cache, we simply invalidate the line from the cache. The last possible case is an exclusive cacheline in the LLC. No special operation is required since the timestamp manager neither has the timestamps nor the data in this case.

The key advantage of this base-delta compression scheme is that all computation is local to each cache without coordination between different caches. This makes the scheme very scalable.

It is possible to extend the base-delta scheme to *wts* and *rts* to further compress the timestamp storage. Specifically, *wts* can be treated as the *bts* and we only need to store the $delta\_rts = rts - wts$ which can be even shorter than $rts - bts$. We defer an evaluation of this extension to future work.

The scheme discussed here does not compress the timestamps over the network and we assume that the network messages still use 64-bit timestamps. It is possible to reduce this overhead by extending the base-delta scheme over the network but this requires coordination amongst multiple caches. We did not implement this extension in order to keep the basic protocol simple and straightforward.

C. Private Write

According to Table II, writing to a cacheline in exclusive state updates both *wts* and *pts* to $Max(pts, rts+1)$. If the core keeps writing to the same address, the *pts* will keep increasing causing other cachelines to expire. If the updated cacheline is completely private to the updating thread, however, there is actually no need to increase the timestamps in order to achieve sequential consistency. According to our definition of global memory order (Definition 1), we can use physical time to order these operations implicitly without increasing the *pts*.

Specifically, when a core writes to a cacheline, the *modified bit* will be set. For future writes to the same cacheline, if the bit is set, then the *pts*, *wts* and *rts* are just set to $Max(pts, rts)$. This means that *pts* will not increase if the line is repeatedly written to. The optimization will significantly reduce the rate at which timestamps increase if most of the accesses from a core are to thread private data.

This optimization does not violate sequential consistency because these writes with the same timestamp are ordered correctly in the physical time order and thus they are properly ordered in the global memory order.

D. Extension: Exclusive and Owned States

In this paper, *MSI* has been used as the baseline directory coherence protocol. *MSI* is the simplest protocol and optimized ones require **E** (Exclusive) and/or **O** (Owned) states. The resulting protocols are *MESI*, *MOSI* and *MOESI*. In this section, we show that Tardis is compatible with both *E* and *O* states.

Similar to the *M* state, the *E* state allows the cacheline to be exclusively cached upon a SH_REQ if no other sharers exist. The core having the cacheline can update the data silently without notifying the directory. In the directory, *M* or *E* cachelines are handled in the same way; an invalidation is sent to the core exclusively caching the line if another core requests it. In Tardis, we can support the *E* state by always returning a cacheline in exclusive state if no other cores seem to be caching it. Note that even if other cores are sharing the line, it can still be returned to the requester in exclusive state. The argument for this is similar to the write after shared argument in Section III-C2; i.e., the lines shared and the line exclusively cached have different ranges of valid timestamps. However, this may not be best for performance. Therefore, we would like to return a cacheline in exclusive state only if the cacheline *seems* to be private. We can add an extra bit for each cacheline indicating whether any core has accessed it since it was put into the LLC. And if the bit is unset, the requesting core gets the line in exclusive state, else in shared state. *E* states will reduce the number of renewals required since cachelines in *E* state will not expire.

The *O* state allows a cacheline to be dirty but shared in the private caches. Upon receiving the WB_REQ request, instead of writing the data back to the LLC or DRAM, the core can change the cacheline to *O* state and directly forward the data to the requesting core. In Tardis, the *O* state can also be supported by keeping track of the owner at the timestamp manager. SH_REQs to the timestamp manager are forwarded to the owner which does cache-to-cache data transfers. Similar to the basic Tardis protocol, when the owner is evicted from the private cache, the cacheline is written back to the LLC or the DRAM and its state in the timestamp manager is changed to Shared or Invalid.

E. Extension: Remote Word Access

Traditionally, a core loads a cacheline into the private cache before accessing the data. But it is also possible to access the data remotely without caching it. Remote word access has been studied in the context of locality-aware directory coherence [18]. Remote atomic operation has been implemented on Tilera processors [19], [20]. Allowing data accesses or computations to happen remotely can reduce the coherence messages and thus improve performance [21].

However, it is not easy to maintain the performance gain of these remote operations with directory coherence under TSO or sequential consistency. For a remote load operation (which might be part of a remote atomic operation), it is not very easy to determine its global memory order since it is hard to know the physical time when the load operation actually happens. As a result, integration of remote access with directory coherence is possible but fairly involved [22].

Consider the example program in Listing 1 where all memory requests are remote accesses. If all requests are issued simultaneously, then both loads may be executed before both stores and the program outputs $A = B = 0$. It is not easy to detect this violation in a directory protocol since we do not know when each memory operation happens. As a result,

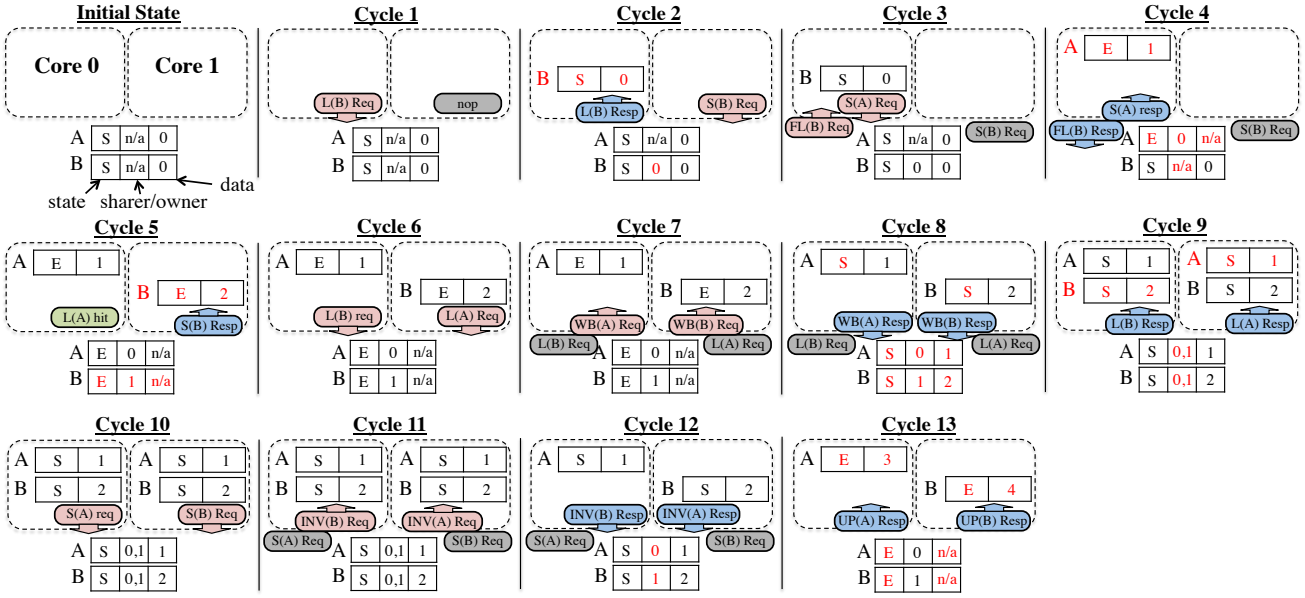


Fig. 2. Execution of the case study program with a directory coherence protocol.

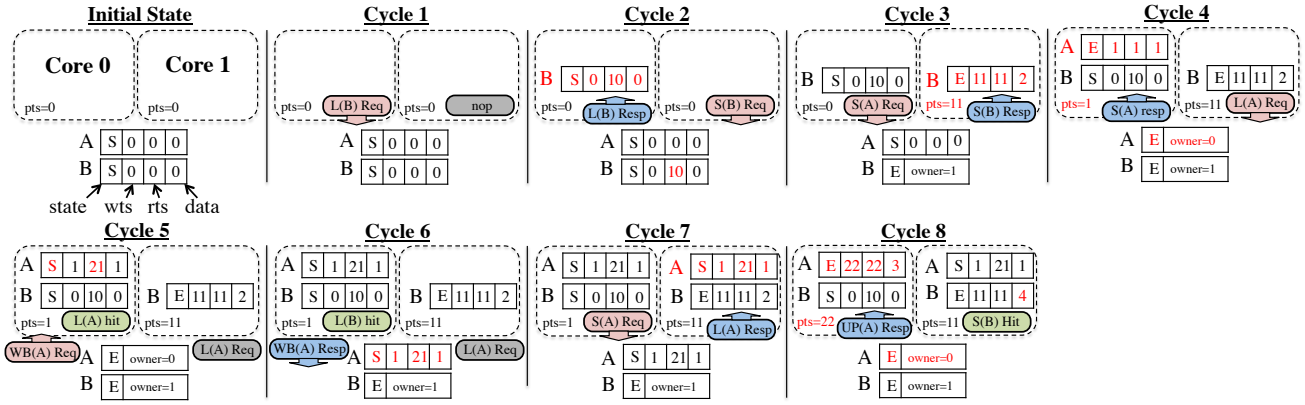


Fig. 3. Execution of the case study program with Tardis protocol.

either the remote accesses are sequentially issued or additional mechanisms need to be added [22].

In Tardis, however, memory operations are ordered through timestamps. It is very easy to determine the memory order for a remote access since it is simply the timestamp of the operation. In Tardis, multiple remote accesses can be issued in parallel and the order can be checked after they return. If any load violates the memory order, it can be reissued with the updated timestamp information (similar to timestamp checking in an out-of-order core).

F. Other Extensions

Atomic operations in Tardis can be implemented the same way as in directory protocols. Tardis can be extended to relaxed consistency models such as Total Store Order (TSO) implemented in Intel x86 processors [23]. Tardis can work with both private Last Level Cache (LLC) or shared LLC.

V. Case Study

In this section, we use an example parallel program as a case study to compare Tardis with an MSI directory protocol.

A. Example

Listing 2 shows the parallel program we use for the case study. In this program, the two cores issue loads and stores to addresses `A` and `B`. The `nop` in Core 1 means that the core spends that cycle without accessing the memory subsystem. The program we use here is a contrived example to highlight the difference between Tardis and a directory coherence protocol.

Listing 2. The case study parallel program

```
[Core 0]        [Core 1]
 L(B)            nop
 A = 1           B = 2
 L(A)            L(A)
 L(B)            B = 4
 A = 3
```

Fig. 2 shows the execution of the program on a directory coherence protocol and Fig. 3 shows how it is executed on Tardis. A cacheline is either in shared (*S*) or exclusive (*E*) state. For Tardis, a static lease of 10 is used. Initially, all private caches are empty and all timestamps are 0. We will explain step by step how Tardis executes the program and highlight the differences between Tardis and the directory protocol.

Cycle 1 and 2: Core 0 sends a shared request to address *B* in cycle 1, and receives the response in cycle 2. The cacheline is reserved till timestamp 10. Core 1 sends an exclusive request to address *B* at cycle 2. In these two cycles, both the directory protocol and Tardis have the same network messages sent and received.

Cycle 3: In Tardis, the exclusive request from core 1 sees that address *B* is shared till timestamp 10. The exclusive ownership is instantly returned and the store is performed at timestamp 11. In the directory protocol, however, an invalidation must be sent to Core 0 to invalidate address *B*. As a result, the exclusive response is delayed to cycle 5. At this cycle, core 0 sends an exclusive request to address *A*.

Cycle 4: In both Tardis and the directory protocol, address *A*'s exclusive ownership can be instantly returned to core 0 since no core is sharing it. The *pts* of core 0 becomes 1 after performing the store. Core 1 performs a shared request to address *A* which needs to get the latest data from core 0 through write back. So the shared response returns in cycle 7. The same *L(A)* instruction in the directory protocol incurs the same latency and network traffic from cycle 6 to 9.

Cycle 5 and 6: In cycle 5, the *L(A)* instruction in core 0 hits in the private cache and thus no request is sent. Also in core 0, the write back request increases address *A*'s *rts* to 21 since the requester's (core 1) *pts* is 11 and the lease is 10. In cycle 6, the *L(B)* instruction in core 0 hits in the private cache since the *pts* is 1 and the cached address *B* is valid till timestamp 10. In the directory protocol, the same *L(B)* instruction is also issued at cycle 6. However, it misses in the private cache since the cacheline was already invalidated by core 1 at cycle 4. So a write back request to core 1 needs to be sent and the shared response returns at cycle 9.

Cycle 7 and 8: At cycle 7, core 0 sends an exclusive request to address *A* and core 1 gets the shared response to address *A*. At cycle 8, the exclusive ownership of address *A* is instantly returned to core 0 and the store happens at timestamp 22 (because addresss *A* has been reserved for reading until timestamp 21). In the directory protocol, the same *S(A)* instruction happens at cycle 10 and the shared copy in core 1 must be invalidated before exclusive ownership is given. Therefore, the exclusive response is returned at cycle 13. Also in cycle 8 in Tardis, core 1 stores to address *B*. The store hits in the private cache. In the directory protocol, the same store instruction happens at cycle 10. Since core 0 has a shared copy of address *B*, an invalidation must be sent and the exclusive response is returned at cycle 13.

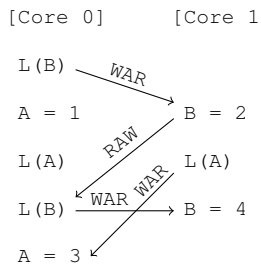
B. Discussion

In this case study, the cycle saving of Tardis mainly comes from the removal of invalidations. While a directory protocol requires that only one version of an address exist at any point in time across all caches, Tardis allows multiple versions to coexist as long as they are accessed at different timestamps.

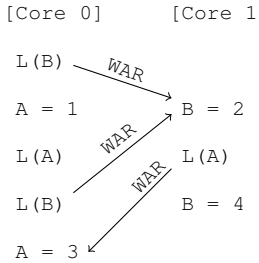
The *pts* in each core shows how Tardis orders the memory operations. At cycle 3, core 1's *pts* jumps to 11. Later at cycle 4, core 0's *pts* jumps to 1. Although the operation from core 0 happens later than the operation from core 1 in physical time, it is the opposite in global memory and physiological time order. Later at cycle 8, core 0's *pts* jumps to 22 and becomes bigger than core 1's *pts*.

In Tardis, a load may still return a old version of an address after it is updated by a different core, as long as sequential consistency is not violated. As a result, Tardis may produce a different instruction interleaving than a directory protocol. Listings 3 and 4 show the instruction interleaving of the directory protocol and Tardis, respectively, on our example program.

Listing 3. Instruction interleaving in directory protocol



Listing 4. Instruction interleaving in Tardis



In the directory protocol, the second *L(B)* instruction from core 0 is between the two stores to address *B* from core 1 in the global memory order. In Tardis, however, the same *L(B)* instruction is ordered before both stores. Such reordering is possible because Tardis enforces sequential consistency in physiological time order which can be different from physical time order.

VI. Evaluation

We now evaluate the performance of Tardis in the context of multi-core processors.

A. Methodology

We use the Graphite [24] multi-core simulator for our experiments. The default hardware parameters are listed in Table V. The simplest directory protocol *MSI* is used as the baseline in this section.[3] This baseline keeps the full sharer information for each cacheline and thus incurs non-scalable storage overhead. To model a more scalable protocol, we use the Ackwise [11] protocol which keeps a limited number of

[3]Other states, e.g., **O** (Owner) and **E** (Exclusive) can be added to an MSI protocol to improve performance; such states can be added to Tardis as well to improve performance as described in Section IV-D.

TABLE V
SYSTEM CONFIGURATION.

System Configuration	
Number of Cores	N = 64 @ 1 GHz
Core Model	In-order, Single-issue
Memory Subsystem	
Cacheline Size	64 bytes
L1 I Cache	16 KB, 4-way
L1 D Cache	32 KB, 4-way
Shared L2 Cache per Core	256 KB, 8-way
DRAM Bandwidth	8 MCs, 10 GB/s per MC
DRAM Latency	100 ns
2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Flit Width	128 bits
Tardis Parameters	
Lease	10
Self Increment Period	100 cache accesses
Delta Timestamp Size	20 bits
L1 Rebase Overhead	128 ns
L2 Rebase Overhead	1024 ns

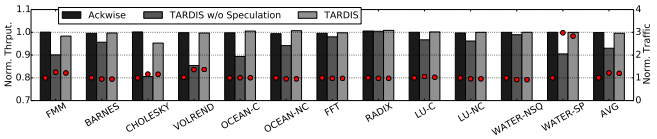


Fig. 4. Performance of Tardis at 64 cores. Both throughput (bars) and network traffic (dots) are normalized to baseline MSI.

sharers and broadcasts invalidations to all cores when the number of sharers exceeds the limit.

In our simulation mode, Graphite includes functional correctness checks, where the simulation fails, e.g., if wrong values are read. All the benchmarks we evaluated in this section completed which corresponds to a level of validation of Tardis and its Graphite implementation. Formal verification of Tardis can be found in [13].

Splash-2 [25] benchmarks are used for performance evaluation. For each experiment, we report both the throughput (in bars) and network traffic (in red dots).

1) Tardis Configurations: Table V also shows the default Tardis configuration. For load requests, a static lease of 10 has been chosen. The *pts* at each core self increments by one for every 100 cache accesses (self increment period). The Base-delta compression scheme is applied with 20-bit delta timestamp size and 64-bit base timestamp size. When the timestamp rolls over, the rebase overhead is 128 ns in L1 and 1024 ns in an LLC slice.

Static lease and self increment period are chosen in this paper for simplicity – both parameters can be dynamically changed for better performance based on the data access pattern. Exploration of such techniques is left for future work.

B. Main Results

1) Throughput: Fig. 4 shows the throughput of Ackwise and Tardis on 64 in-order cores, normalized to baseline MSI. For Tardis, we also show the performance with speculation turned off. For most benchmarks, Tardis achieves similar performance compared to the directory baselines. On average, the performance of Tardis is within 0.5% of the baseline MSI and Ackwise.

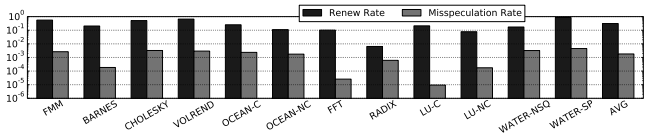


Fig. 5. Renew and misspeculation rate of Tardis at 64 cores. Y-axis is in log scale.

If the speculation is turned off, Tardis's performance becomes 7% worse than MSI. In this case, the core stalls while waiting for the renewal, in contrast to the default Tardis where the core reads the value speculatively and continues execution. Since most renewals are successful, speculation hides a significant amount of latency and makes a big difference in performance.

2) Network Traffic: The red dots in Fig. 4 show the network traffic of Ackwise and Tardis normalized to the baseline MSI. On average, Tardis with and without speculation incurs 19.4% and 21.2% more network traffic. Most of this traffic comes from renewals. Fig. 5 shows the percentage of renew requests and missspeculations out of all LLC accesses. Note that the y-axis is in log scale.

In benchmarks with lots of synchronizations (e.g., `CHOLESKY`, `VOLREND`), cachelines in the private cache frequently expire generating a large number of renewals. In `VOLREND`, for example, 65.8% of LLC requests are renew requests which is 2× of normal LLC requests. As discussed in Section III-F, a successful renewal only requires a single flit message which is cheaper than a normal LLC access. So the relative network traffic overhead is small (36.9% in `VOLREND` compared to baseline MSI).

An outlier is `WATER-SP`, where Tardis increases the network traffic by 3×. This benchmark has very low L1 miss rate and thus very low network utilization. Even though Tardis incurs 3× more traffic, the absolute amount of traffic is still very small.

In many other benchmarks (e.g., `BARNES`, `WATER-NSQ`, etc.), Tardis has less network traffic than baseline MSI. The traffic reduction comes from the elimination of invalidation and cache eviction traffic.

From Fig. 5, we see that the misspeculation rate for Tardis is very low, less than 1% renewals failed on average. A speculative load is considered a miss if the renew fails and a new version of data is returned. Having a low misspeculation rate indicates that the vast majority of renewals are successful.

3) Timestamp Discussion: Table VI shows how fast the *pts* in a core increases, in terms of the average number of cycles to increase the *pts* by 1. Table VI also shows the percentage of *pts* increasing caused by self increment (cf. Section III-E).

Over all the benchmarks, *pts* is incremented by 1 every 263 cycles. For a delta timestamp size of 20 bits, it rolls over every 0.28 seconds. In comparison, the rebase overhead (128 ns in L1 and 1 μs in L2) becomes negligible. This result also indicates that timestamps in Tardis increase very slowly. This is because they can only be increased from accessing shared read/write cachelines or self increment.

On average, 26.6% of timestamp increasing is caused by self

TABLE VI
TIMESTAMP STATISTICS

Benchmarks	Ts. Incr. Rate (cycle / timestamp)	Self Incr. Perc.
FMM	322	22.5%
BARNES	155	33.7%
CHOLESKY	146	35.6%
VOLREND	121	23.6%
OCEAN-C	81	7.0%
OCEAN-NC	85	5.6%
FFT	699	88.5%
RADIX	639	59.3%
LU-C	422	1.4%
LU-NC	61	0.1%
WATER-NSQ	73	12.8%
WATER-SP	363	29.1%
AVG	263	26.6%

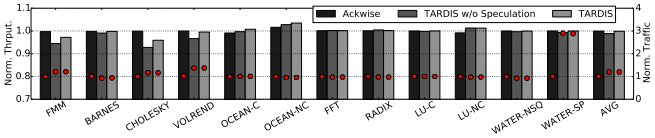


Fig. 6. Performance of Tardis on 64 out-of-order cores.

increment and the percentage can be as high as 88.5% (FFT). This has negative impact on performance and network traffic since unnecessarily increasing timestamps causes increased expiration and renewals. Better livelock avoidance algorithms can resolve this issue; we leave this for future work.

C. Sensitivity Study

1) In-order vs. Out-of-Order Core: Fig. 6 shows the performance of Tardis on out-of-order cores. Compared to in-order cores (Fig. 4), the performance impact of speculation is much smaller. When a renew request is outstanding, an out-of-order core is able to execute independent instructions even if it does not speculate. As a result, the renewal's latency can still be hidden. On average, Tardis with and without speculation is 0.2% and 1.2% within the performance of baseline MSI respectively.

The normalized traffic of Tardis on out-of-order cores is not much different from in-order cores. This is because both core models follow sequential consistency and the timestamps assigned to the memory operations are virtually identical. As a result, the same amount of renewals is generated.

2) Self Increment Period: As discussed in Section III-E, we periodically increment the *pts* at each core to avoid livelock. The *self increment period* specifies the number of data cache accesses before self incrementing the *pts* by one. If the period is too small, the *pts* increases too fast causing more expirations; more renewals will be generated which increases network traffic and hurts performance. Fast growing *pts*'s also overflow the *wts* and *rts* more frequently (*cf.* Section VI-C4) which also hurts performance. If the period is too large, however, an update at a remote core may not be observed locally quickly enough, which degrades performance.

Fig. 7 shows the performance of Tardis with different self increment period. The performance of most benchmarks is not sensitive to this parameter. In FMM and CHOLESKY, performance goes down when the period is 1000. This is

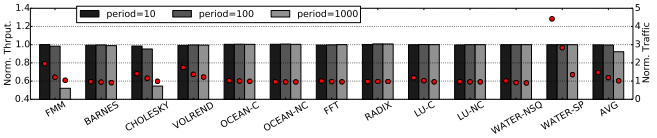
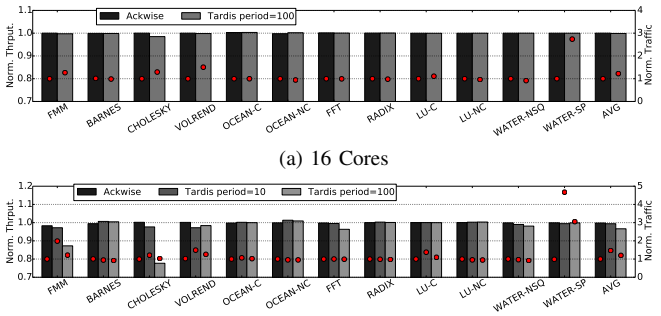


Fig. 7. Performance of Tardis with different self increment period.



(a) 16 Cores

(b) 256 Cores

Fig. 8. Performance of Tardis on 16 and 256 cores.

because these two benchmarks heavily use spinning (busy waiting) to synchronize between threads. If the period is too large, the core spends a long time spinning on the stale value in the private cache and cannot make forward progress.

Having a larger self increment period always reduces the total network traffic because of fewer renewals. Given the same performance, a larger period should be preferred due to network traffic reduction. Our default self increment period is 100 which has reasonable performance and network traffic.

Ideally, the self increment period should dynamically adjust to the program's behavior. For example, the period can be smaller during spinning but larger for the rest of the program where there is no need to synchronize. Exploration of such schemes is deferred to future work.

3) Scalability: Fig. 8 shows the performance of Tardis on 16 and 256 cores respectively.

At 16 cores, the same configurations are used as at 64 cores. On average, the throughput is within 0.2% of baseline MSI and the network traffic is 22.4% more than the baseline MSI.

At 256 cores, two Tardis configurations are shown with self increment period 10 and 100. For most benchmarks, both Tardis configurations achieve similar performance. For FMM, CHOLESKY, however, performance is worse when the period is set to 100. As discussed in Section VI-C2, both benchmarks heavily rely on spinning for synchronization. At 256 cores, spinning becomes the system bottleneck and period $= 100$ significantly delays the spinning core from observing the updated variable. It is generally considered bad practice to heavily use spinning at high core count.

On average, Tardis with period $= 100$ performs 3.4% worse than MSI with 19.9% more network traffic. Tardis with period $= 10$ makes the performance 0.6% within baseline MSI with 46.7% traffic overhead.

Scalable storage is one advantage of Tardis over directory protocols. Table VII shows the per cacheline storage overhead in the LLC for two directory baselines and Tardis. Full-map MSI requires one bit for each core in the system, which is $O(N)$ bits per cacheline. Both Ackwise and Tardis can achieve

TABLE VII
STORAGE OVERHEAD OF DIFFERENT COHERENCE PROTOCOLS (BITS PER LLC CACHELINE) WITH 4 SHARERS FOR ACKWISE AT 16/64 AND 8 SHARERS AT 256 CORES.

# cores (N)	full-map MSI	Ackwise	Tardis
16	16 bits	16 bits	40 bits
64	64 bits	24 bits	40 bits
256	256 bits	64 bits	40 bits

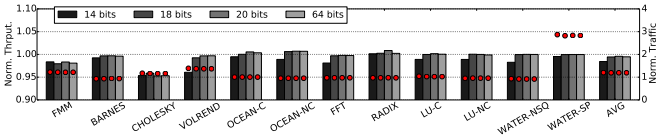


Fig. 9. Performance of Tardis with different timestamp size.

$O(\log N)$ storage but Ackwise requires broadcasting support and is thus more complicated to implement.

Different from directory protocols, Tardis also requires timestamp storage for each L1 cacheline. But the per cacheline storage overhead does not increase with the number of cores.

4) Timestamp Size: Fig. 9 shows Tardis's performance with different timestamp sizes. All numbers are normalized to the baseline MSI. As discussed in Section IV-B, short timestamps roll over more frequently, which degrades performance due to the rebase overhead. According to the results, at 64 cores, 20-bit timestamps can achieve almost the same performance as 64-bit timestamps (which never roll over in practice).

5) Lease: Finally, we sweep the lease in Fig. 10. Similar to the self increment period, the lease controls when a cacheline expires in the L1 cache. Roughly speaking, a large lease is equivalent to a long self increment period. For benchmarks using a lot of spinning, performance degrades since an update is deferred longer. The network traffic also goes down as the lease increases. For most benchmarks, however, performance is not sensitive to the choice of lease. However, we believe that intelligently choosing leases can appreciably improve performance; for example, data that is read-only can be given an infinite lease and will never require renewal. We defer the exploration of intelligent leasing to future work.

VII. RELATED WORK

We discuss related works on timestamp based coherence protocols (Section VII-A) and scalable directory coherence protocols (Section VII-B).

A. Timestamp based coherence

To the best of our knowledge, none of the existing timestamp based coherence protocols is as simple as Tardis and achieves the same level of performance as Tardis. In all of these protocols, the timestamp notion is either tightly coupled with physical time, or these protocols rely on broadcast or snooping for invalidation.

Using timestamps for coherence has been explored in both software [26] and hardware [27]. TSO-CC [28] proposed a hardware coherence protocol based on timestamps. However, it only works for the TSO consistency model, requires broadcasting support and frequently self-invalidates data in private caches. The protocol is also more complex than Tardis.

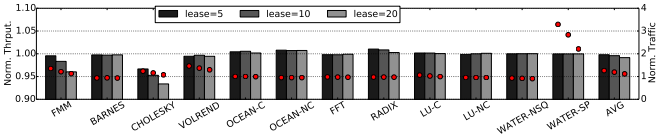


Fig. 10. Performance of Tardis with different lease.

In the literature we studied, Library Cache Coherence (LCC) [9] is the closest algorithm to Tardis. Different from Tardis, LCC uses the physical time as timestamps and requires a globally synchronized clock. LCC has bad performance because a write to a shared variable in LCC needs to wait for all the shared copies to expire which may take a long time. This is much more expensive than Tardis which only updates a counter without any waiting. Singh et al. used a variant of LCC on GPUs with performance optimizations [10]. However, the algorithm only works efficiently for release consistency and not sequential consistency.

Timestamps have also been used for verifying directory coherence protocols [29], for ordering network messages in a snoopy coherence protocol [30], and to build write-through coherence protocols [31], [32]. None of these works built coherence protocols purely based on timestamps. Similar to our work, Martin et. al [30] give a scheme where processor and memory nodes process coherence transactions in the same logical order, but not necessarily in the same physical time order. The network assigns each transaction a logical timestamp and then broadcasts it to all processor and memory nodes without regard for order, and the network is required to meet logical time deadlines. Tardis requires neither broadcast nor network guarantees. The protocol of [31] requires maintaining absolute time across the different processors, and the protocol of [32] assumes isotach networks [33], where all messages travel the same logical distance in the same logical time.

B. Scalable directory coherence

Some previous works have proposed techniques to make directory coherence more scalable. Limited directory schemes (e.g., [34]) only track a small number of sharers and rely on broadcasting [11] or invalidations when the number of sharers exceeds a threshold. Although only $O(\log N)$ storage is required per cacheline, these schemes incur performance overhead and/or require broadcasting which is not a scalable mechanism.

Other schemes have proposed to store the sharer information in a chain [35] or hierarchical structures [36]. Hierarchical directories reduce the storage overhead by storing the sharer information as a k-level structure with $\log_k N$ bits at each level. The protocol needs to access multiple places for each directory access and thus is more complex and harder to verify.

Previous works have also proposed the use of coarse vectors [37], sparse directory [37], software support [38] or disciplined programs [39] for scalable coherence. Recently, some cache coherence protocols have been proposed for 1000-core processors [40], [12]. These schemes are directory based and require complex hardware/software support. In contrast, Tardis can achieve similar performance with a very simple

protocol.

VIII. Conclusion

We proposed a new memory coherence protocol, Tardis, in this paper. Tardis is directly derived from the sequential consistency model. Compared to popular directory coherence protocols, Tardis is simpler to implement and validate, and has better scalability. Tardis matches the baseline directory protocol in performance in the benchmarks we evaluated. For these reasons, we believe Tardis to be a competitive coherence protocol for future massive-core and large-scale shared memory systems.

References

[1] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *Computers, IEEE Transactions on*, vol. 100, no. 12, pp. 1112–1118, 1978.

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

[3] "Tile-gx family of multicore processors," http://www.tilera.com.

[4] Intel, "Intel Xeon Phi Coprocessor System Software Developers Guide," 2014.

[5] 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*. IEEE, 2010, pp. 1–6.

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

[7] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.

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

[9] 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*. IEEE, 2011, pp. 1–8.

[10] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 578–590.

[11] 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.

[12] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012, pp. 1–12.

[13] X. Yu, M. Vijayaraghavan, and S. Devadas, "A Proof of Correctness for the Tardis Cache Coherence Protocol," *CoRR*, vol. abs/1505.06459, May 2015. [Online]. Available: http://arxiv.org/abs/1505.06459

[14] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.

[15] D. L. Weaver and T. Germond, "The SPARC Architecture Manual," 1994.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[17] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," in *In Proceedings of the 1991 International Conference on Parallel Processing*, 1991, pp. 355–364.

[18] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 523–534.

[19] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 33–48.

[20] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming," in *High Performance Embedded Architectures and Compilers*. Springer, 2010, pp. 3–17.

[21] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 209–220, 2014.

[22] G. Kurian, "Locality-aware Cache Hierarchy Management for Multicore Processors," Ph.D. dissertation, Massachusetts Institute of Technology, 2014.

[23] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, 2010.

[24] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *International Symposium on High-Performance Computer Architecture*, 2010.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *International Symposium on Computer Architecture*, 1995.

[26] S. L. Min and J.-L. Baer, "A timestamp-based cache coherence scheme." Citeseer, 1989.

[27] S. Nandy and R. Narayan, "An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems." Citeseer, 1994.

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

[29] 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*. ACM, 1998, pp. 67–76.

[30] M. M. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood, "Timestamp snooping: an approach for extending SMPs," *ACM SIGOPS Operating Systems Review*, vol. 34, no. 5, pp. 25–36, 2000.

[31] R. Bisiani, A. Nowatzyk, and M. Ravishankar, "Coherent Shared Memory on a Distributed Memory Machine," in *In Proc. of the 1989 Int'l Conf. on Parallel Processing (ICPP'89)*, 1989, pp. 133–141.

[32] C. Williams, P. F. Reynolds, and B. R. de Supinski, "Delta Coherence Protocols," *IEEE Concurrency*, vol. 8, no. 3, pp. 23–29, Jul. 2000.

[33] P. F. R. Jr., C. Williams, and R. R. W. Jr., "Isotach Networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 4, pp. 337–348, 1997.

[34] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *25 years of the international symposia on Computer architecture (selected papers)*. ACM, 1998, pp. 353–362.

[35] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, vol. 23, no. 6, pp. 49–58, 1990.

[36] Y.-C. Maa, D. K. Pradhan, and D. Thiebaut, "Two economical directory schemes for large-scale cache coherent multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 19, no. 5, p. 10, 1991.

[37] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *International Conference on Parallel Processing*, 1990.

[38] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, 1991, pp. 224–234.

[39] 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," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 155–166.

[40] J. H. Kelm, M. R. Johnson, S. S. Lumetta, and S. J. Patel, "WAYPOINT: scaling coherence to thousand-core architectures," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 99–110.