Tardis 2.0

Citation: Yu, Xiangyao, et al. Tardis 2.0: "Optimized Time Traveling Coherence for Relaxed Consistency Models." PACT '16 Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, 11-15 September, 2016, Haifa, Israel, ACM Press, 2016, pp. 261–74.

As Published: http://dx.doi.org/10.1145/2967938.2967942

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/115327

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models

Xiangyao Yu
MIT CSAIL
yxy@mit.edu

Hongzhe Liu
Algonquin Regional High School
liuhenry4428@gmail.com

Ethan Zou
Lexington High School
zouethan20@gmail.com

Srinivas Devadas
MIT CSAIL
devadas@mit.edu

ABSTRACT

Cache coherence scalability is a big challenge in shared memory systems. Traditional protocols do not scale due to the storage and traffic overhead of cache invalidation. Tardis, a recently proposed coherence protocol, removes cache invalidation using logical timestamps and achieves excellent scalability. The original Tardis protocol, however, only supports the Sequential Consistency (SC) memory model, limiting its applicability. Tardis also incurs extra network traffic on some benchmarks due to renew messages, and has suboptimal performance when the program uses spinning to communicate between threads.

In this paper, we address these downsides of Tardis protocol and make it significantly more practical. Specifically, we discuss the architectural, memory system and protocol changes required in order to implement the TSO consistency model on Tardis, and prove that the modified protocol satisfies TSO. We also describe modifications for Partial Store Order (PSO) and Release Consistency (RC). Finally, we propose optimizations for better leasing policies and to handle program spinning. On a set of benchmarks, optimized Tardis improves on a full-map directory protocol in the metrics of performance, storage and network traffic, while being simpler to implement.

1. INTRODUCTION

As the number of cores on a single chip increases, the cache coherence protocol becomes a potential scalability and performance bottleneck. *Snoopy* coherence protocols [1] work well for small-scale systems with a few cores, but do not scale due to the traffic pressure caused by broadcasting messages on the bus. *Directory-based* coherence protocols [2, 3] have better scalability and are widely used in multicore processors today [4, 5]. For future systems with hundreds or even thousands of cores, however, the storage overhead of a full-map directory becomes a serious scalability bottleneck. A number of enhancements to directory coherence protocols have been proposed in the literature [6, 7, 8, 9] to improve scalability. These enhancements typically sacrifice performance and incur extra implementation and verification complexity.

As an alternative, a recently proposed cache coherence protocol called *Tardis* shows better scalability while maintaining simplicity and high performance [10]. The key insight behind Tardis is that it is sufficient but *unnecessary* to enforce the global memory order in physical time order, which is what traditional coherence protocols do. Instead, Tardis uses a combination of logical timestamps and physical time. In Tardis, a memory write does not invalidate shared copies in other cores' private caches. Instead, the write is immediately performed at a logical timestamp greater than the timestamps of all the shared copies. The new data version can coexist with old versions at the same physical time but is ordered after the old versions in logical time.

Introducing logical timestamps into the coherence protocol brings several salient properties to Tardis. First, Tardis is *scalable*; it has no sharer list and only requires $O(\log N)$ storage per cacheline for an N-core system. It also does not require broadcasting support. Tardis is also *simple*; it uses timestamps to explicitly express the consistency model and is easy to reason about and verify [11]. Different from other protocols, Tardis does not need invalidation and is therefore able to achieve better performance on some benchmarks.

There are three drawbacks of the original Tardis protocol. The first drawback is that it only supports the *sequential consistency* (SC) memory model. While SC is simple and well studied, commercial processors usually implement more relaxed models. Intel x86 [12] and SPARC [13] processors can support *Total Store Order* (TSO); ARM [14] and IBM Power [15] processors implement weaker consistency models. It is difficult for commercial processors to adopt Tardis if the target memory models are not supported.

Another drawback of Tardis is the renew message that is used to extend the logical lease of a shared cacheline in a private cache. These messages incur extra latency and bandwidth overhead. Finally, Tardis uses a timestamp self-increment strategy to avoid livelock. This strategy has suboptimal performance when threads communicate via spinning.

In this paper, we will address these drawbacks of the original Tardis protocol and make it more practical. Specifically, we will discuss the changes to the cores and the memory subsystem in order to implement TSO and other relaxed consistency models on Tardis. A formal proof that our algorithm correctly implements TSO is also given. We also propose new optimization techniques (MESI, livelock detection and lease prediction) to reduce the number of renew messages in Tardis, which improves performance.

Our simulations over a wide range of benchmarks indicate that our optimizations improve the performance and reduce the network traffic of Tardis. Compared to a full-map directory-based coherence protocol at 64 cores, optimized Tardis is able to achieve better performance (1.1% average improvement, up to 9.8%) and lower network traffic (2.9% average reduction, up to 12.4%) at the same

PACT '16, September 11-15, 2016, Haifa, Israel
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967942

time. The improvement increases as the system scales to 256 cores where Tardis improves performance (average 4.7%, upto 36.8%) and reduces the network traffic (average 2.6%, upto 14.6%). While the optimized and baseline Tardis protocols require timestamps in the L1 cache, they are, overall, more space-efficient than full-map directory protocols and simpler to implement.

2. BACKGROUND

In this section, we introduce the key concepts behind the basic Tardis protocol (Section 2.1 and Section 2.2) and the protocol itself (Section 2.3 and Section 2.4). We describe the pros and cons of Tardis vis-à-vis a directory coherence protocol (Section 2.5).

2.1 Physiological Time

Most consistency models define a global order for all memory loads and stores. Traditional cache coherence protocols enforce this global memory order using physical time order. This requires an invalidation mechanism for write-after-read data dependency. Specifically, a write to a shared cacheline has to invalidate all the shared copies of that line because the write must be ordered after all previous reads with respect to physical time. The invalidation mechanism requires either broadcasting support in the network (as in snoopy coherence protocols) or a per cacheline sharer list in the directory (as in directory-based coherence protocols) and is therefore not able to scale to large numbers of cores.

Tardis avoids invalidation by introducing logical timestamps into the cache coherence protocol and enforcing the global memory order in both logical and physical time order. In Tardis, a write does not invalidate shared copies, instead, a new version of the cacheline is created at a later logical time at which point all previous shared copies have expired. This usage of logical time to order data bears some similarity to Lamport clock [16] and the timestamp ordering [17, 18] concept in database concurrency control algorithms.

Specifically, Tardis invented a new concept of time called *physiological time* and enforces the global memory order with respect to it. Using $<_{pt}$ and $<_{ts}$ to represent physical time and logical timestamp order respectively, the physiological time order ($<_{pl}$) is defined by Eq. (1).

$$X <_{pl} Y := X <_{ts} Y \text{ or } (X =_{ts} Y \text{ and } X <_{pt} Y) \qquad (1)$$

An operation X is before Y in physiological time order if X has a smaller timestamp or if they have the same timestamp but X happens earlier in physical time. If two operations happen at the same logical and physical time, then we can use any metric to break the tie (since they do not have a conflict). Physiological time makes it easy to reason about consistency models implemented on Tardis.

2.2 Sequential Consistency (SC)

The original Tardis protocol [10] implements sequential consistency, which requires that *"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) appear in this sequence in the order specified by its program"* [19].

Sequential consistency can also be defined using the following two axiomatic rules, where X and Y are two memory operations, $<_p$ and $<_m$ are the program and global memory order respectively, and $L(a)$ and $S(a)$ are load and store to address a respectively.

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

Rule 2: Value of $L(a) =$ Value of $\text{Max}_{<_m} \{S(a) \mid S(a) <_m L(a)\}$

Rule 1 says that the sequential order should agree with the program order and Rule 2 says that each load should observe the preceding store in the sequential order.

In Tardis, the global memory order ($<_m$) in Rule 1 of SC can be implemented using the physiological time order ($<_{pl}$). Section 2.3 describes in detail how Tardis implements SC.

2.3 Tardis with SC

This section presents the baseline Tardis protocol proposed in [10]. For simplicity, we assume a multicore architecture with a private L1 cache per core and a shared L2 as the Last Level Cache (LLC). We also assume a single-threaded in-order core model where an instruction can be issued only after the previous instruction has finished its execution and committed. In practice, Tardis can be implemented on multi-threaded out-of-order cores as well. The physical time when an instruction commits is its *physical commit time*. Each instruction also has a *logical commit timestamp* indicating the logical time when it commits. For the rest of the paper, the term "timestamp" always indicates logical time. Also, when we say an operation is performed at a certain physical/logical time, we mean the physical/logical commit time of that operation.

2.3.1 Timestamp Management

Although both physical and logical commit time are used for reasoning about Tardis. Physical commit time is not explicitly stored in hardware and thus there is no need to read from CPU clocks. This is an important property which makes Tardis scalable. In Tardis, physical time order is meaningful only for operations that already happen in physical time order, e.g., instructions committed from the same core, or a load observing a preceding store. Orders between other operations are enforced using logical timestamps.

Logical timestamps in Tardis are explicitly stored and managed in hardware. For SC, each thread maintains a program timestamp (*pts*) as the logical commit timestamp of the last operation in the program order. For multithreaded cores, each hardware context maintains its own *pts* in hardware. According to Rule 1 in the definition of SC, the global memory order (i.e., physiological commit time order in Tardis) must agree with the program order. Therefore, the *pts* should monotonically increase with the program order[1].

Each cacheline in Tardis maintains a write timestamp (*wts*) and a read timestamp (*rts*). *wts* is the commit timestamp of the last store and *rts* can be considered the commit timestamp of the last load. Together they mean that the value is valid between *wts* and *rts* in logical time. Without further optimizations, for a cache with (*wts*, *rts*), a load can only be performed at *ts* such that $wts \leq ts \leq rts$, and a store can only be performed at *ts* such that $ts \geq rts + 1$. In terms of logical time, this means that a load always observes the preceding store and only one version of data exists at any timestamp.

2.3.2 Tardis-SC Protocol

We now present the baseline Tardis-SC protocol. For clarity, we first explain how shared caching is implemented in Tardis-SC and then discuss exclusive caching.

Shared Caching

An important concept in Tardis is *timestamp leasing* of a cacheline. In order to load a shared copy into a core's private cache, the core needs to get a lease on the cacheline. Multiple cores can cache the same cacheline in shared state with overlapping leases. But an update can be performed on the cacheline only after all shared leases expire. Leasing in Tardis is implemented with respect to logical but not physical time.

In private caches, the *rts* of a cacheline indicates the end of the lease on the line. A load operation can be performed as long as $pts \leq rts$ and the load updates *pts* to *Max(pts, wts)*. If $pts > rts$,

[1] So for every $X <_p Y$, we have $X \leq_{ts} Y$ and $X <_{pt} Y$ (due to in-order processing) and therefore $X <_{pl} Y$.

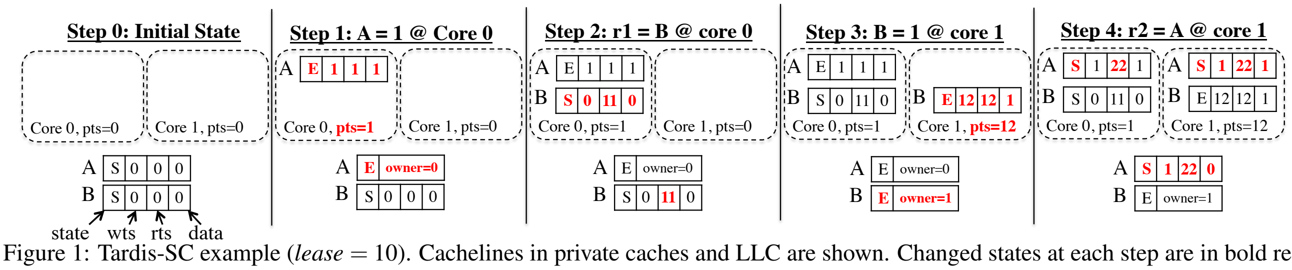


Figure 1: Tardis-SC example (*lease* $= 10$). Cachelines in private caches and LLC are shown. Changed states at each step are in bold red.

however, the load cannot be directly performed (for example, the line may have been modified in the LLC after *rts* but before *pts* in logical time) and a renew request has to be sent to the LLC to extend the *rts* of the data version. If the version in the LLC is the same as the version in the L1 (i.e., they have the same *wts*), the extension is successful and the extended *rts* is returned. Otherwise, the extension fails and the latest version is returned instead. But regardless, the returned *rts* will be greater than *pts* and the load can now be performed. The baseline Tardis uses speculative loading to hide the latency of renew messages.

In the shared LLC, the *rts* of a cacheline is the maximal *rts* among all the shared copies of the same address, namely, the end of all shared leases. A load or renew request from a core to the LLC may extend the cacheline's *rts* if the requesting core has a sufficiently large *pts*. Specifically, the *rts* is updated to *Max(rts, pts + lease)* for the load request where *lease* is a constant (e.g., *lease* = 10) in the baseline Tardis. The *lease* is chosen to be non-zero so that the line will be valid in the L1 for longer logical time and therefore not be constantly renewed by the core. The data and both *wts* and *rts* are returned to the requesting core in the response.

In Tardis, if a core keeps reading a stale cacheline at a *pts* smaller than the cacheline's *rts*, the core may never observe a later update and livelock may occur. To guarantee that a write is eventually observed by other cores, Tardis forces the *pts* of each core to periodically increase so that a stale version of a cacheline will eventually expire. We return to the subject of livelock in Section 5.2.

Exclusive Caching

Exclusive caching in Tardis is similar to the directory baseline for the most part. For example, a write operation needs to acquire exclusive ownership from the LLC; ownership transfers if a another core writes to the same cacheline; if another core reads the cacheline, the owner downgrades the line to shared state and writes the data back to the LLC. For a write operation at an L1, the newly created version has *new_version.wts* = *previous_version.rts + 1*.

When a core writes to a cacheline that is shared by one or more cores, however, Tardis has fundamentally different behavior than a traditional cache coherence protocol. When the write request reaches a shared cacheline in the LLC, unlike traditional cache coherence protocols, Tardis does not send invalidations to the sharers. Instead, exclusive ownership can be immediately returned to the requesting core. The core can perform the write at a *wts* greater than the cacheline's current *rts*. Since all the shared copies expire at this *rts*, this current write operation happens at a timestamp greater than all reads to the previous version. Therefore, the write is after those reads in the global memory order. Given the definition of SC (Rule 2 in particular), this behavior is sequentially consistent according to physiological time order.

2.4 Tardis-SC Example

Listing 1: Example Program

```
    initially A = B = 0
 [Core 0]            [Core 1]
  A = 1               B = 1
  r1 = B              r2 = A
```

Fig. 1 shows how the dual-thread program in Listing 1 is executed in Tardis-SC. For simplicity, the example executes the four operations one at a time, and executes the operations from core 0 before those from core 1. Initially, both cores' L1s are empty; all cachelines in the LLC are in shared state; and all timestamps are 0.

Step 1: Core 0 stores to address A. A store request is sent to the LLC. Since A is in shared state in the LLC, exclusive ownership is immediately returned to core 0. Core 0 performs the store at timestamp $A.rts + 1 = 1$. After the store, core 0's *pts* becomes 1. The new version of A has $wts = rts = 1$.

Step 2: Core 0 loads from address B. A load request is sent to the LLC with $pts = 1$. B's *rts* in the LLC is extended to $Max(rts, pts + lease) = 11$. The data value and both timestamps are returned to Core 0's L1 cache.

Step 3: Core 1 stores to address B. A store request is sent to the LLC. Since B is in shared state, exclusive ownership is immediately returned to core 1, which performs the store at timestamp $A.rts + 1 = 12$ and updates the *pts*, *wts* and *rts*. Note that although Core 0 is caching the previous version of B, no invalidation is sent for Core 1's store. After the store is performed, there are actually two different versions of B coexisting in the core 0's and core 1's L1 caches. However, they are valid at different logical timestamps. Core 0's copy is valid from timestamp 0 to 11, while core 1's copy is valid from timestamp 12. Therefore, sequential consistency is still maintained in physiological time order.

Step 4: Core 1 loads from address A. A load request is sent to the LLC. Since Core 0 is the current owner, a message is sent to Core 0 asking for data and *wts*/*rts* writeback. The LLC is updated based on the response and a shared copy is returned to Core 1. Since the *pts* of the load request is 12, the *rts* of all A's cachelines are extended to $pts + lease = 22$.

2.5 Tardis vs. Directory-based Coherence

A key advantage of Tardis compared to a traditional physical time based coherence protocol is the removal of the invalidation mechanism. This simplifies protocol design, reduces storage overhead, and can potentially achieve better performance.

Removing the invalidation mechanism means that the LLC does not notify the sharing cores when a cacheline is modified. So each core should contact the LLC to learn the freshness of a cacheline. This is done through the renew messages (cf. Section 2.3). In a sense, replacing invalidations with renewals is replacing a push-based model with a poll-based model. A poll-based model is usually simpler and more scalable, but may lead to wasted messages. For example, read only data may be constantly renewed due to expiration, but these messages would not exist in an invalidation-based protocol. The latency and bandwidth overhead of the extra renew traffic is the major potential disadvantage of Tardis compared to directory coherence protocols.

Some optimization techniques (e.g., speculative read) have been proposed to hide the latency of renewals [10]. However, the performance and network traffic overhead can still be significant for some benchmarks. We will present our solutions to the renewal problem in Section 5.

3. TARDIS WITH TSO

The original Tardis protocol only supports the sequential consistency (SC) memory model. Although SC is intuitive and precisely defined, it may overly constrain the ordering of memory operations. In practice, this may lead to suboptimal performance. To resolve this disadvantage of SC, relaxed consistency models have been proposed and widely implemented in real systems. Most of these models focus on the relaxation of the program order in SC (Rule 1 in the SC definition). Specifically, the program order of a core may appear out-of-order in the global memory order. The more relaxed a model is, the more flexibility it has to reorder operations, which usually leads to better overall performance.

In this section, we show how Tardis can be generalized to relaxed consistency models. We first use Total Store Order (TSO) as a case study since it has a precise definition and is the most widely adopted. We will present the formal definition of TSO (Section 3.1), the Tardis-TSO protocol (Section 3.2), an example program (Section 3.3) and optimizations (Section 3.4). Finally, we generalize the discussion to other memory models (Section 3.5).

3.1 Formal Definition of TSO

The TSO consistency model relaxed the Store → Load constraint in the program order. This allows a load after a store in the program order to be flipped in the global memory order (assuming that the load and the store have no data or control dependency). In an out-of-order processor, this means that an outstanding store does not block the following loads. When a store reaches the head of the *Re-Order Buffer* (ROB), it can retire to the *store buffer* and finish the rest of the store operation there. The loads following the store can therefore commit early before the store is actually done. Since store misses are common in real applications, this relaxation can lead to significant performance improvement.

Similar to SC, the definition of TSO also requires a global order (specified using $<_m$) of all memory instructions. However, the global memory order does not need to follow the program order for Store → Load dependency. Specifically, TSO can be defined using the following three rules [20]. The differences between TSO and SC are highlighted in boldface.

Rule 1: L(a) $<_p$ L(b) $\Rightarrow$ L(a) $<_m$ L(b)
L(a) $<_p$ S(b) $\Rightarrow$ L(a) $<_m$ S(b)
S(a) $<_p$ S(b) $\Rightarrow$ S(a) $<_m$ S(b)
~~**S(a) $<_p$ L(b) $\Rightarrow$ S(a) $<_m$ L(b)**~~

Rule 2: Value of $L(a)$ = Value of Max$_{<m}$ $\{S(a) \mid S(a) <_m L(a)$ **or S(a) $<_p$ L(a)**$\}$

Rule 3: **X $<_p$ FENCE $\Rightarrow$ X $<_m$ FENCE**
FENCE $<_p$ X $\Rightarrow$ FENCE $<_m$ X

In TSO, the program order implies the global memory order only for Load → Load, Load → Store and Store → Store constraints. Since there is no Store → Load constraint, a load can bypass the pending store requests and commit earlier (Rule 1). Although the load is after the store in the program order, it is before the store in the global memory order.

In an out-of-order processor, TSO can be implemented using a store buffer which is a FIFO for pending store requests that have retired from ROB. If the address of a load exists in the store buffer, then the value in the store buffer is directly returned; otherwise, the load accesses the memory hierarchy (Rule 2).

TSO uses a *fence* instruction when a Store → Load order needs to be enforced (Rule 3). In a processor, a fence flushes the store buffer enforcing that all previous stores have finished so that a later committed load is ordered after stores before the fence in physical time. If all memory operations are also fences, then TSO becomes SC.

3.2 Tardis-TSO Protocol

In this section, we describe how TSO can be implemented on Tardis. Specifically, we discuss the changes to the timestamp management policy as compared to the Tardis SC protocol.

3.2.1 Program Timestamp Management

The original Tardis SC protocol uses a single program timestamp (*pts*) to represent the commit timestamp of an instruction. Since the program order always agrees with the global memory order in SC, *pts* monotonically increases in the program order.

In TSO, however, the program order does not always agree with the global memory order. Following Rule 1 in TSO's definition, a store's timestamp is no less than the timestamps of all preceding loads, stores and fences in the program order. A load's timestamp is no less than the timestamps of all preceding loads and fences, but not necessarily preceding stores. As a result, a single monotonically increasing *pts* is insufficient to represent the ordering constraint.

To express the different constraints for loads and stores respectively, we split the original *pts* into two timestamps. The *store timestamp* (*sts*) represents the commit timestamp of the last store, and the *load timestamp* (*lts*) represents the commit timestamp of the last load. Like *pts*, both *sts* and *lts* are maintained in each core in hardware. According to Rule 1, both should monotonically increase in the program order because of the Load → Load and Store → Store constraints. Furthermore, the timestamp of a store (*sts*) should be no less than the timestamp of the preceding load (*lts*) because of the Load → Store constraint. For a load, however, its *lts* can be smaller than *sts* because there is no Store → Load constraint.

A fence can be simply implemented as a synchronization point between *sts* and *lts*. Specifically, a fence sets *lts* = *Max(lts, sts)*. This enforces Rule 3 in TSO because operations after the fence are ordered after operations before the fence in physiological time order (and therefore the global memory order). If each memory operation is also a fence, then the commit timestamp for each operation monotonically increases and the protocol becomes Tardis SC.

In a traditional coherence protocol, the main advantage of TSO over SC is the performance gain due to loads bypassing stores in the store buffer. In Tardis, besides bypassing, TSO can also reduce the number of renewals compared to SC. This is because the *lts*/*sts* in TSO may increase slower compared to the *pts* in SC. As a result, fewer shared cachelines expire.

3.2.2 Data Timestamp Management

The timestamp management logic largely remains the same when the consistency model switches from SC to TSO. However, the timestamp rules for data in the store buffer need some slight changes. For single-threaded cores, timestamp management in the private L1 can also be changed for load requests for potentially better performance.

To load a cacheline in L1 that is not *dirty* (i.e., the data has not been changed by the core since it was cached), the timestamp rule is exactly the same as in SC, i.e., the *lts* should fall within the lease of the cacheline ($wts \leq lts \leq rts$); *lts* jumps to *wts* if *lts* is smaller than *wts* and a renewal is sent if *lts* is greater than *rts*.

For a dirty cacheline in the store buffer or L1 cache, however, the *lts* does not have to be greater than the *wts* of the cacheline. With respect to the global memory order, this means that the load can commit at an *lts* smaller than the commit timestamp of the store

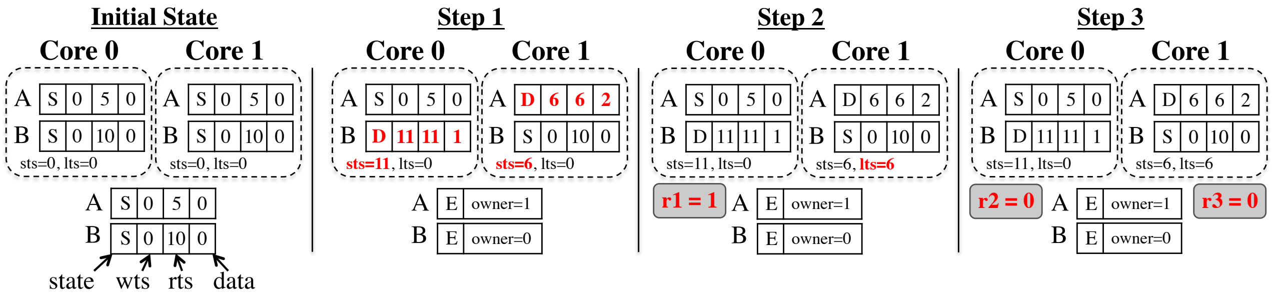


Figure 2: Tardis-TSO example (*lease* = 10). A private cacheline in *D* state means it is modified and dirty. Changed states at each step are highlighted in red.

creating the data (*wts*). This behavior certainly violates SC but it is completely legal in TSO.

According to Rule 2 of TSO, a load should return either the last store in global memory order or the last store in program order, depending on which one has a larger physiological time. Since a dirty cacheline was written by a store from the current core prior to the load, even if the load has a smaller commit timestamp than the store, Rule 2 still holds. A more formal proof of correctness is presented in Section 4.

Unlike in traditional processors, TSO can be implemented with Tardis even on in-order cores that do not have a store buffer. This is because Tardis can figure out the correct memory ordering using logical timestamps as will become clear in the example presented in the next section. We note that our implementation of Tardis TSO still has a store buffer for better performance, but it is not required for functional correctness.

Note that if multiple threads can access the same private cache, then the above optimization for dirty L1 cachelines may not be directly applied in L1 cache (but it is still applied in the store buffer). Because a dirty line might be written by any thread sharing the L1. For these systems, this optimization can be turned off in the L1.

3.3 TSO Example

Listing 2: Example Program

```
[core0]                 [core1]
 B = 1                  A = 2
 L(B) → r1              FENCE
 L(A) → r2              L(B) → r3
```

We use the example program in Listing 2 to demonstrate how timestamps are managed in Tardis TSO. The execution of the program is shown in Fig. 2. For simplicity, we do not model the store buffer and execute one instruction per step for each core.

Initially, both addresses A and B are cached in Shared (S) state in both cores' private caches as well as the shared LLC. *wts* of all lines are 0; *rts* of all lines of address A are 5 and *rts* of all lines of address B are 10.

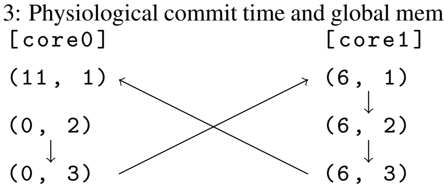
Step 1: core 0 writes to address B and core 1 writes to address A. Exclusive ownership of A and B are given to core 1 and core 0 respectively, and both stores are performed by jumping ahead in logical time to the end of the lease. After the stores, core 0's *sts* jumps to timestamp 11 and core 1's *sts* jumps to 6, but the *lts* of both cores remain 0.

Step 2: core 0 loads address B. The value of the previous store from core 0 is returned ($r1 = 1$). Since B is in the dirty state, the load does not increase the *lts* (Section 3.2.2). In core 1, a fence instruction is executed which synchronizes the *lts* and *sts* to timestamp 6.

Step 3: core 0 loads address A. Since its *lts* is 0 which falls between the *wts* and *rts* of cacheline A, this is a cache hit and value 0 is returned ($r2 = 0$). In core 1, the load to address B also hits the L1 since its *lts* = 6 falls within B's lease. As a result, the loaded value is also 0 ($r3 = 0$).

Listing 3 shows the physiological commit time for each instruction in Listing 2. It also shows the global memory order using arrows. Physiological time is represented using a logical timestamp and physical time pair (ts, pt) where ts is the commit timestamp and pt is the physical commit time of the instruction. According to the definition, $(ts_1, pt_1) < (ts_2, pt_2)$ if $ts_1 < ts_2$ or ($ts_1 = ts_2$ and $pt_1 < pt_2$).

Listing 3: Physiological commit time and global memory order.



```
[core0]                   [core1]
(11, 1)                   (6, 1)
                            ↓
(0, 2)                    (6, 2)
  ↓                         ↓
(0, 3)                    (6, 3)
```

The execution is definitely not sequentially consistent since the program order in core 0 is violated between `B = 1` and `L(B)`. But it obeys all the invariants of TSO. Note that the store buffer is not included in the example since we are modeling in-order cores, but TSO can still be implemented. This feature is not available in traditional physical time based coherence protocols. For this example, adding the store buffer will not change the hardware behavior.

3.4 TSO Optimizations

Many optimization techniques have been proposed in the literature to improve performance of the basic SC/TSO consistency model. Examples include load speculation to hide Load → Load and Store → Load dependency, and store prefetch to enhance Store → Store and Load → Store performance [21]. For TSO, the speculation can go over fences.

Tardis TSO is compatible with these optimizations. In fact, it may be even simpler to support them on Tardis than on traditional coherence protocols since the timestamps can help preserve/check memory order. For example, for load → load relaxation, multiple loads can be speculatively executed in parallel, and the *wts* and *rts* of the loaded cachelines are stored inside the core (e.g., in the ROB). To enforce load → load dependency, the processor only needs to commit instructions with ascending timestamp order (and reissue a load with a new timestamp if necessary). In contrast, a traditional processor needs to snoop on invalidation messages in order to detect a speculation failure. Fence speculation can also be implemented in a similar way using timestamps. In general, Tardis allows all memory operations to be speculatively executed arbitrarily, as long as their commit timestamps obey the consistency model. This flexibility makes it easier to reason about and implement these optimizations.

Table 1: Memory order constraints for different consistency models. *L* for load, *S* for store and *F* for fence. For release consistency, *rel* for release and *acq* for acquire.

Consistency Model	Ordinary Orderings	Synchronization Orderings
SC	L → L, L → S, S → L, S → S	
TSO	L → L, L → S, S → S	S → F, F → L, F → F
PSO	L → L, L → S	S → F, F → S, F → L, F → F
RC		L/S → rel, acq → L/S, rel/acq → rel/acq

3.5 Other Relaxed Consistency Models

Similar to TSO, other memory consistency models (Table 1) can also be supported in Tardis with proper changes to the timestamp rules. Given the relationship between the program order and the global memory order, it is usually straightforward to adapt Tardis for different models. In this section, we briefly discuss Partial Store Order (PSO) and Release Consistency (RC) to illustrate how Tardis can adapt to them with minimal algorithmic change.

3.5.1 Partial Store Order (PSO)

The PSO consistency model [13] relaxes both the Store → Load and the Store → Store orderings. Similar to TSO, we use the *lts* and *sts* to model the program order constraints. In PSO, since Load → Load is enforced but Store → Load is not, which is the same as TSO, the rule for *lts* is also the same as TSO. Namely, *pts* should monotonically increase independently of store timestamps.

The timestamp order for stores, however, does not need to monotonically increase, since Store → Store is relaxed. Therefore, the timestamp of a store (*ts*) only needs to be no less than the *lts* ($ts \geq lts$). And *sts* represents the largest store timestamp so far (instead of the last store timestamp), namely $sts = Max(sts, ts)$.

For a fence instruction, *lts* synchronizes with *sts*, namely $lts = Max(lts, sts)$. The resulting *lts* is the timestamp of the fence.

3.5.2 Release Consistency (RC)

Release consistency [22] relaxes all the program order constraints; furthermore, it also relaxes the ordering constraints for synchronizations. Specifically, an *acquire* guarantees that all the following (but not the previous) operations are ordered after the acquire and a *release* guarantees that all the previous (but not the following) operations are ordered before the release.

In Tardis, we need to maintain timestamps for acquire (*acquire_ts*) and release (*release_ts*) operations, as well as the maximal commit timestamp (*max_ts*) so far. A normal load or store operation (commit timestamp *ts*) can be performed as long as its timestamp is greater than *acquire_ts* (*ts* ≥ *acquire_ts*); *max_ts* represents the largest commit timestamp as seen by the core so far (*max_ts* = *Max*(*max_ts*, *ts*)). At a release instruction, *release_ts* and *max_ts* are synchronized (*release_ts* = *Max*(*release_ts*, *max_ts*)). At an acquire instruction, *acquire_ts* and *release_ts* are synchronized (*acquire_ts* = *Max*(*acquire_ts*, *release_ts*)).

4. PROOF OF CORRECTNESS

In this section, we prove that the algorithm in Section 3.2 correctly implements TSO. We first introduce several definitions and invariants (in the form of lemmas) of the Tardis protocol (Section 4.1). We then use these lemmas to prove that our algorithm implements TSO (Section 4.2).

4.1 Invariants of Tardis protocol

The invariants of Tardis introduced in this section are true regardless of the memory model used by the system. While some of the lemmas come from a proof of Tardis SC ([11]), we restate them in the language of physiological time.

To simplify the discussion, we first introduce the concept of *master* and *snapshot* cachelines.

DEFINITION 1 (MASTER, SNAPSHOT CACHELINE). *A cacheline is a master cacheline if it is in M state in an L1, or in S state in the LLC. A cacheline is a snapshot cacheline if it is in S state in an L1.*

Here are some facts about master and snapshot cachelines:

Fact 1. *For each cacheline with (wts, rts), the data value comes from a previous store that committed at logical time wts.*

In Tardis, the only way to change *wts* of a cacheline is by performing a store to it and the new *wts* after the store always equals the commit timestamp of the store.

Fact 2. *For each address in the system, at most one master cacheline exists but multiple snapshot cachelines may exist.*

This is similar to the single-writer, multiple-reader (SWMR) invariant in traditional coherence protocols. In Tardis, however, multiple snapshot cachelines (i.e., shared L1 cachelines) can coexist with a master cacheline (e.g., exclusive L1 cacheline) at the same physical time.

Fact 3. *A snapshot cacheline is derived by taking a snapshot of data and timestamps of a master cacheline*.

In the protocol, an L1 shared cacheline may be derived from a shared response from the LLC or from downgrading a modified cacheline in an L1. In both cases, it is a snapshot of a master cacheline.

For the proof, we use (*ts*, *pt*) to represent the physiological time of an operation, where *ts* and *pt* are the logical and physical commit time of the operation, respectively.

LEMMA 1. *For each address, the wts and rts of the master cachelines never decrease.*

PROOF. *In the basic Tardis protocol, no operation decreases the timestamp of a cacheline. For a master cacheline, its wts/rts can only increase (through writes) or stay the same (through exclusive response to L1 or write back to LLC) but never decrease.* □

LEMMA 2. *For a master cacheline, no store to the address has happened at* (wts', wpt') *such that* $(wts, wpt) < (wts', wpt')$*, where the (wts, wpt) is the physiological commit time of the store that created the data in the cacheline.*

PROOF. *If such a store has ever happened, it would have created a version of master cacheline with write timestamp* $wts' > wts$*. However, Lemma 1 states that the wts of the master cachelines do not decrease. This contradicts the fact that the wts of the cacheline is currently less than* wts'*.* □

LEMMA 3. *For a snapshot cacheline (with wts and rts) at physical time pt, no store has happened at* (ts, pt') *such that* $(wts, wpt) < (ts, pt') < (rts, pt)$*, where the (wts, wpt) is the physiological commit time of the store that created the data in the cacheline.*

PROOF. *A snapshot cacheline is always copied from a master cacheline. When the snapshot is taken, according to Lemma 2, no store to the address exists at a physiological time after* (wts, wpt)*. Since then, according to Tardis rules, a store can only happen at a logical timestamp greater than rts of the cacheline. Therefore, any later store has a greater physiological time than* (rts, ∞)*. So no store can exist between* (wts, wpt) *and* (rts, pt)*.* □

4.2 Tardis TSO Proof

We first make the following assumption about a processor implementing TSO.

ASSUMPTION 1. *A TSO processor commits instructions in physical time order if their global memory order has to follow the program order according to Rule 1 and 3 of the TSO definition. Specifically,*

$L(a) <_p L(b) \Rightarrow L(a) <_{pt} L(b)$
$L(a) <_p S(b) \Rightarrow L(a) <_{pt} S(b)$
$S(a) <_p S(b) \Rightarrow S(a) <_{pt} S(b)$
$X <_p FENCE \Rightarrow X <_{pt} FENCE$
$FENCE <_p X \Rightarrow FENCE <_{pt} X$

This assumption does not require extra hardware changes to existing processors. In fact, processors implementing TSO today already follow this assumption.

THEOREM 1. *The protocol of Section 3.2 implements TSO.*

PROOF. *We will prove that each rule in the definition of TSO is maintained in our protocol.*

Proof for Rule 1. *According to Section 3.2.1, both lts and sts monotonically increase and for a store, its sts is no less than the current lts. In other words,*

$L(a) <_p L(b) \Rightarrow L(a) \leq_{ts} L(b)$
$L(a) <_p S(b) \Rightarrow L(a) \leq_{ts} S(b)$
$S(a) <_p S(b) \Rightarrow S(a) \leq_{ts} S(b)$

Combined with our assumption of a TSO processor (Assumption 1), we have:

$X \leq_{ts} Y$ *and* $X <_{pt} Y$
$\Rightarrow X <_{ts} Y$ *or* $(X =_{ts} Y$ *and* $X <_{pt} Y)$
$\Rightarrow X <_{pl} Y$
$\Rightarrow X <_m Y$

which finishes the proof of Rule 1 of TSO.

Proof for Rule 2. *We will prove that for each memory load, the returned value is the one specified by Rule 2 of the TSO definition. According to Fact 1, for each cacheline in Tardis, the data value was created by a store that happened at the cacheline's wts. A load to a cacheline observes the value of this particular store. Assuming that the store committed at physical time wpt, we need to prove that the physiological time of this store, which is (wts, wpt), is greater than all the other stores in the set* $S = S_1 \cup S_2$ *where* $S_1 = \{S(a) | S(a) <_m L(a)\}$ *and* $S_2 = \{S(a) | S(a) <_p L(a)\}$ *and that the observed store is also in this set. Specifically, we consider the following two cases.*

Case 1, load to a shared L1 cacheline. Timestamps of the cacheline are wts and rts. Because $wts \leq lts \leq rts$ *and* $wpt < pt$*, the observed store must be in the set* S_1*. According to Lemma 3, at physical time pt, no store has happened at* (wts', wpt') *such that* $(wts, wpt) < (wts', wpt') < (rts, pt)$*. As a result, the observed store has the largest physiological time in set* S_1*.*

We now prove that all stores in set S_2 *have smaller physiological time than the observed store. This can be proven by contradiction. If such a store has been executed by the current core at* (wts', wpt') $> (wts, wpt)$*, the current core must have owned a master cacheline with* wts' *after the store. Since the master cacheline's wts never decreases and a snapshot cacheline is copied from a master cacheline, the current shared cacheline must have wts greater than* wts'*, contradicting the assumption. So the observed store has the largest physiological time in set* $S_1 \cup S_2$*, proving the invariant.*

Case 2, load to a modified L1 cacheline. In this case, the loaded cacheline is a master cacheline. If the cacheline is not dirty, then $wts \leq lts$*. So the observed store is in* S_1*. If the cacheline is dirty, then the observed store must be before the current load in the program order and is therefore in* S_2*. According to Lemma 2, no store has happened to the address at* $(wts', wpt') > (wts, wpt)$*. So the observed store must have the largest physiological time in set* $S = S_1 \cup S_2$*.*

Proof for Invariant 3. *In Tardis TSO, a fence synchronizes lts and sts. Tardis TSO enforces the following:*

$X <_p FENCE \Rightarrow X \leq_{ts} FENCE$
$X <_p FENCE \Rightarrow X <_{pt} FENCE$
$FENCE <_p X \Rightarrow FENCE \leq_{ts} X$
$FENCE <_p X \Rightarrow FENCE <_{pt} X$

Combining these equations and applying the definition of physiological time, we prove the last invariant of TSO. □

5. RENEWAL REDUCTION TECHNIQUES

As discussed in Section 2, a major drawback of the original Tardis protocol is the renewal problem. With load speculation, latency of renew messages can be largely hidden, but network traffic overhead remains. A renew message is unnecessary if the renewed line is unchanged. In this section, we discuss techniques to reduce unnecessary renew messages.

5.1 MESI on Tardis

The original Tardis protocol implements MSI where a read to a private cacheline loads it in S (shared) state in the L1 cache. As the *lts* (or *pts* in SC) increases in a core, these data will expire and be renewed. These renewals are unnecessary since there is no need to maintain coherence for a core's private data.

The MESI protocol can resolve this issue. MESI adds an E (Exclusive) state to MSI. The E state is granted for a request if the cacheline is not shared by other cores. Like a cacheline in M state, an exclusive cacheline in a private cache never expires. If the *lts* is greater than the *rts*, the *rts* of that line can be increased to *lts* with no renewal. This can be done since the line is not shared by other cores, namely, a *master cacheline* according to Definition 1. Therefore, renewal does not happen for private data in MESI.

Different from traditional coherence protocols, Tardis can grant E state to a core even if other cores are still sharing the line. This is similar to granting M state without the need of invalidation. However, for performance reasons, it is still desirable to only grant E state for private data. In Tardis, a cacheline is likely to be not shared if it has just been loaded from DRAM, and if it has just been downgraded from E/M to S state. Therefore, we add an E-bit to each cacheline in the LLC to indicate whether the cacheline is likely to be shared or not. The E-bit is set when the cacheline is loaded from DRAM and also when the cacheline is downgraded to S state. The E-bit is reset when the cacheline becomes cached upon load request. Note that E-bit may be unset even if no core is sharing the line (e.g., all sharers silently evict the line). This does not affect the functional correctness of the implementation.

5.2 Livelock Detection

A disadvantage of Tardis is that propagating a store to other cores may take an arbitrarily long time. This is because the writer does not notify the sharing cores who may not *pull* the latest value if they keep reading the stale version. In the worst case if a core spins on a stale cacheline with a small *lts* (or *pts* in SC), it never sees the latest update and livelock occurs (Listing 4). Although such livelock is not precluded by the consistency model, it should be disallowed by the coherence protocol which requires that every write should eventually propagate to other cores. In practice, this spinning behavior is commonly used for communication in parallel programs.

Listing 4: Threads communicate through spinning

```
 [Core 0]                [Core 1]
//spin (lts = 10)       //store (sts = 20)
while (!done) {         ...
  sleep(1)              done = true
}                       ...
```

5.2.1 Baseline: Periodic Self Increment

The original Tardis protocol solves the livelock problem by self incrementing the *pts* (or *lts* in TSO) periodically to force the logical time in each core to move forward. For a spinning core (e.g., core 0 in Listing 4), the *lts* will increase and eventually become greater than the *rts* of the cacheline being spun on at which point the line expires and the latest value will be loaded.

Frequently self incrementing *lts* causes two performance issues. First, the shared cachelines in the private cache may frequently expire, generating renewals. For programs without spinning, these renewals are unnecessary but incur network traffic and latency. Second, for cachelines that have an *rts* much larger than the current *lts* of the core, it may take a significant amount of time before the *lts* increases to *rts* for the stale line to expire.

5.2.2 Livelock Detector

We make a key observation that a *check* message can be sent to check the freshness of a cacheline even before the cacheline actually expires. Like a renew request, if the latest data is newer than the cached data, the latest cacheline is returned. If the cached data is already the latest version, however, a check response is returned without extending the *rts* of the cacheline in the LLC.

The check request can resolve both drawbacks of the self increment scheme. Since a core does not need to frequently increase its *lts*, the number of renewals can be reduced. Also, a check request can be sent when *lts* is much smaller than *rts*, so a core does not need to wait a long time for the cacheline to expire.

Generally, a check request should be sent when the program seems to livelock as it keeps loading stale cachelines. In practical programs, such a livelock is usually associated with variable spinning, which typically involves a small number of cachelines and is therefore easy to detect in hardware.

Algorithm 1: Livelock Detection Algorithm (called for each read request to a shared L1 cacheline).

```
Input: addr    // memory access address
Return Value: whether to issue check request
Internal State: AHB, thresh_count

if AHB.contains(addr) then
    AHB[addr].access_count ++
    if AHB[addr].access_count == thresh_count then
        AHB[addr].access_count = 0
        return true
    end if
else
    AHB.enqueue(addr)
    AHB[addr].access_count = 0
    return false
end if
```

We designed a small piece of hardware next to each core to detect livelock. It contains an *Address History Buffer* (AHB) and a threshold counter (*thresh_count*). The AHB is a circular buffer keeping track of the most recently loaded addresses. Each entry in AHB contains the address of a memory access, and an *access_count*, which is the number of accesses to the address since it was loaded to AHB. When *access_count* becomes greater than the *thresh_count*, a check request is sent for this address (Algorithm 1). The value of *thresh_count* can be static or dynamic. We chose to use an adaptive threshold counter scheme (Algorithm 2) in order to minimize the number of unnecessary check messages.

Algorithm 2: Adaptive Threshold Counter Algorithm (called for each check response).

```
Input: check_update // whether the checked address has been
    updated.
Internal State: thresh_count
Constant: min_count, max_count, check_count,
    check_thresh

if check_update then
    thresh_count = min_count
    check_count = 0
else
    check_count ++
    if check_count == check_thresh and
    thresh_count < max_count then
        thresh_count = thresh_count × 2
    end if
end if
```

The livelock detection algorithm (Algorithm 1) is executed when reading a shared cacheline. It is not called when accessing cachelines in E or M state since no livelock can occur for those accesses. If the accessed address does not exist in the AHB, a new entry is allocated. Since AHB is a circular buffer, this may evict an old entry from it. We use the LRU replacement policy here but other replacement policies should work equally well. For an AHB hit, the *access_count* is incremented by 1. If the counter saturates (i.e., reaches *thresh_count*), a check request is sent and the *access_count* is reset. All *access_counts* are reset to 0 when the *lts* increases due to a memory access, since this indicates that the core is not livelocking, and thus there is no need to send checks.

The counter *thresh_count* may be updated for each check response (Algorithm 2). If the checked address was updated by another core, then *thresh_count* should decrease to the minimal value, indicating that check requests should be sent more frequently since data seems to be updated frequently. Otherwise, if *check_thresh* number of consecutive check requests returned without data being changed, then *thresh_count* is doubled since it appears unnecessary to send check requests that often. Adaptively determining the value of *thresh_count* can reduce the number of unnecessary check requests if a thread needs to spin for a long time before the data is updated.

Note that the livelock detector can only detect spinning involving loads to less than *M* (the number of entries in AHB) distinct addresses. So in theory, the livelock detector cannot capture all possible livelocks and self incrementing *lts* is still required to guarantee forward progress. For practical programs, however, spinning typically involves a small number of distinct addresses. So the livelock detector is able to capture livelock in the vast majority of programs. We still self increment *lts* periodically but the frequency can be much lower, since most programs' performance does not rely on this mechanism anymore.

5.3 Lease Prediction

During regular operation of Tardis, memory stores are the main reason that the timestamps increase in the system. The amount that an *sts* increases is determined by the lease of the previous data version, because the *sts* of the store must be no less than the cacheline's previous *rts*. Therefore, the lease of each cacheline is important to the timestamp incrementing rate as well as the renew rate. The original Tardis protocol uses a static lease for every shared cacheline. We first show that a static leasing policy may incur unnecessary renewals (Section 5.3.1). We then propose a dynamic leasing policy to mitigate the problem (Section 5.3.2).

5.3.1 Static Lease vs. Dynamic Lease

In the code snippet shown in Listing 5, both cores run the same program. They both load addresses A and B and then store to B. When the cachelines are loaded to L1 caches, they are all reserved with a static lease *L*. When the store to address B is performed, both cores' *sts* jump ahead by at least *L*. At the end of the loop, the FENCE instruction increases *lts* to the value of *sts*.

Listing 5: The case study parallel program

```
[Core 0]                 [Core 1]

 while(B < 10) {         while(B < 10) {
   print A                  print A
   B++                      B++
   FENCE                    FENCE
 }                        }
```

In the next iteration when both cores load A again, the cacheline has expired in the L1 caches. Because the *lts* has already jumped ahead by *L* due to the previous store B and the fence, but the lease on A was also *L*. As a result, in each iteration of the loop, *lts* and *sts* jump ahead by *L* and A is renewed at each core. All these renewals to A are successful, and therefore unnecessary, since A has never been changed. Note that using a larger static *L* does not solve the problem since *lts*/*sts* will jump ahead further and A will still expire.

Our solution to this problem is to use different leases for different addresses. Intuitively, we want to use large leases for read only or read intensive data, and use small leases for write intensive data. In the example in Listing 5, if A has lease 100 and B has lease 10, then each store to B increases the *sts* and *lts* only by 10. So it takes about 10 iterations before A has to be renewed again. The renew rate is mainly a function of the ratio between these two leases; the absolute lease values are not critical.

In a real system, it is non-trivial to decide what data should have a larger or smaller lease. Here, we explore hardware only solutions and design a predictor to decide the lease for each cacheline. It is possible to do this more accurately with software support, such explorations are left for future work.

5.3.2 Lease Predictor

Our lease predictor is based on the observation that cachelines that are frequently renewed are more likely to be read intensive. Therefore, a cacheline should be reserved with a larger lease if it is renewed frequently. The logic to determine the lease value is built into the LLC which has knowledge of all the renewals.

For each renew request from the L1 to the LLC, the last lease (*req_lease*) of the cacheline is also sent to the LLC and is processed by the lease predictor. For an L1 miss, the cacheline has no last lease so the *min_lease* is used. The lease predictor computes a proper lease based on the *req_lease* and the predictor's current internal lease (*cur_lease*). Specifically, the algorithm of our lease predictor is shown in Algorithm 3.

Algorithm 3: Lease Prediction Algorithm (called for each LLC request).

```
Input
    req_lease   // lease in the request
    req_type    // WRITE, READ or RENEW
Return Value: Lease of the returned cacheline.
Internal State: cur_lease

if req_type == WRITE then
    cur_lease = min_lease
else if req_type == RENEW and req_lease == cur_lease
        and cur_lease < max_lease then
    cur_lease = cur_lease × 2
end if
return cur_lease
```

For a write request, the *cur_lease* is updated to the minimal lease value (*min_lease*). Our reasoning is that the write indicates that the cacheline might be write intensive and so assigning a large lease to it may cause unnecessary renewals of other cachelines. For a normal read request, *cur_lease* is used for the requested cacheline. For a renew request, *cur_lease* is compared with the request lease (*req_lease*). If they are different, then *cur_lease* is used for the cacheline. Otherwise, *cur_lease* is doubled since the cacheline seems to be renewed multiple times by the same core and is therefore likely to be read intensive. If *cur_lease* already reached the maximal value (*max_lease*), then it should no longer increase.

The initial value of *cur_lease* is the minimal value (*min_lease*). This means that for a cacheline first loaded to the LLC, we always assume it to be write intensive. We made this design decision because incorrectly giving a large lease to a write intensive cacheline is much more harmful than giving a small lease to a read intensive cacheline. If a cacheline with a large lease is written, a large number of cachelines in the core's L1 might expire due to the program timestamp jumping ahead. In contrast, if a read only cacheline is given a small lease, only this cacheline needs to be renewed and other cachelines are not affected.

6. EVALUATIONS

We evaluate the performance of Tardis with relaxed consistency models and the optimizations proposed in Section 5.

6.1 Methodology

6.1.1 System Configuration

We use the Graphite [23] multicore simulator to model the Tardis coherence protocol. Configurations of the hardware, Tardis, and its enhancements are shown in Table 2 and Table 3.

For the baseline Tardis, we implemented all the optimizations in the original Tardis protocol, including speculative reads when a cacheline expires in the L1 cache and not incrementing *sts* for private writes [10]. The static lease always equals 8. And the *lts* self increments by 1 for every 100 memory accesses.

For the livelock detector, the address history buffer (AHB) contains 8 entries by default. The threshold counter can take values ranging from 100 to 800. The threshold counter is doubled if 10 consecutive checks respond that the data has not been changed.

The minimal lease is chosen to be 8 and the maximum lease is 64. There are four possible lease values (i.e., 8, 16, 32, 64) in the system. We chose four lease values because having more values does not significantly affect performance.

6.1.2 Baselines

The following coherence protocols are implemented and evaluated for comparison. Except in Section 6.2, all the configurations use the TSO consistency model and MESI.

Directory: Full-map MESI directory coherence protocol.

Base Tardis: Baseline Tardis [10] where *lts* self increments by 1 for every 100 memory accesses.

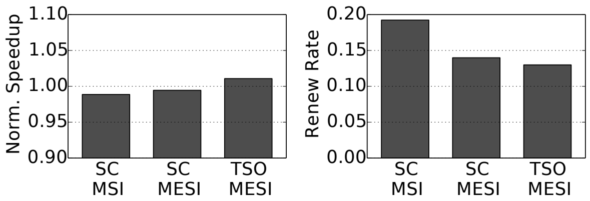
Tardis + live: Baseline Tardis with livelock detection. *lts* self increments by 1 for every 1000 memory accesses.

Table 2: System Configuration.

System Configuration	
Number of Cores	N = 64
Core Model	Out-of-order, 128-entry ROB
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

Table 3: Tardis Configuration.

Baseline Tardis	
Static Lease	8
Self Increment Period	100 memory accesses
Livelock Detector	
AHB size	8 entries
Threshold Counter	min 100, max 800
Check Threshold	10
Lease Prediction	
Minimal Lease Value	8
Maximal Lease Value	64



(a) Speedup (b) Renew Rate

Figure 3: MESI and TSO. Average speedup (normalized to directory + MESI + TSO) and renew rate over all benchmarks.

Tardis + live + lease: Tardis with both a livelock detector and lease predictor.

In Tardis, the base-delta timestamp compression scheme [10] was implemented and each timestamp requires 20 bits of storage. This means that each cacheline requires 40 bits in total for *wts* and *rts*, regardless of the number of cores in the system. The full-map directory protocol, in contrast, requires an N-bit sharer list for each cacheline in the LLC for an N-core system. However, note that timestamps are required in the L1 for the baseline and optimized Tardis unlike in the directory protocol.

Our experiments are executed over 22 benchmarks selected from Splash2 [24], PARSEC [25], sparse linear algebra [26] and OLTP database applications [27]. For sparse linear algebra, we evaluated sparse matrix multiplicaion (SPMV) and symmetric Gauss-Seidel smoother (SYMGS) from the HPCG benchmark suite (Top 500 supercomputer ranking). For OLTP database, we evaluated two benchmarks YCSB and TPCC. All benchmarks are executed to completion.

6.2 Consistency Models and MESI

Fig. 3 shows the speedup of MESI and TSO on Tardis normalized to the baseline directory coherence protocol with MESI and TSO. All experiments have both livelock detection and lease prediction enabled for a fair comparison. Both MESI and TSO can improve the overall performance of Tardis. On average, using MESI instead of MSI improves the performance of Tardis SC by 0.6%; using TSO instead of SC further improves performance by 1.7%.

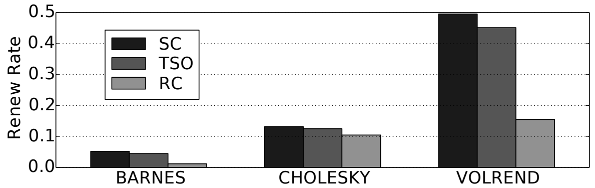


Figure 4: Renew rate of consistency models SC, TSO and RC.

MESI and TSO can also reduce the renew rate of Tardis. We define renew rate as the ratio of the number of renew requests over the total number of LLC accesses. Fig. 3b shows the renew rate reduction of MESI and TSO on Tardis. MESI reduces the number of renew messages since private readonly data is always cached in E state and therefore never renewed. TSO further reduces the renew rate since the *lts* may increase slower than the *pts* in SC (cf. Section 3.2.1) leading to fewer expirations/renewals. MESI and TSO together can reduce the average renew rate from 19.2% to 13.0%.

Although not shown in these figures, TSO can also significantly decrease the rate at which timestamps increase. This is because *lts* can stay behind *sts*. Therefore, *lts* and *sts* may increase slower than how *pts* increases in Tardis SC. On average, the timestamp increment rate in Tardis TSO is 78% of the rate in Tardis SC.

Fig. 4 shows the renew rate of SC, TSO and RC on Tardis with MESI. Due to some issues with running x86 pthread synchronization primitives on RC, we implemented hardware-based synchronization for this experiment and therefore use a separate graph to present the results. For many benchmarks that are *data-race-free*, relaxing the consistency models does not significantly reduce the renew rate. For some other benchmarks (like the ones in Fig. 4), however, more relaxed consistency models lead to significantly fewer renewals.

6.3 Livelock Detector and Lease Predictor

We now evaluate the performance and hardware overhead of the livelock detector and lease predictor of Section 5. All coherence protocols in this section use MESI and TSO.

6.3.1 Performance and Network Traffic

Fig. 5 shows the performance after adding livelock detection and lease prediction compared to the baseline Tardis protocol. All numbers are normalized to the baseline directory protocol.

First, we see that CHOLESKY and SYMGS on baseline Tardis have much worse performance than the directory protocol. Both benchmarks heavily use spinning to communicate between threads. As a result, it may take a long time for the cacheline spun on to expire. The livelock detector can close the performance gap between Tardis and directory because a spinning core is able to observe the latest data much earlier. RADIOSITY also uses spinning. However, the core does other computation between checking the value of the variable. Therefore, our livelock detector cannot capture such spinning behavior and forward progress is enforced through self incrementing *lts*. This leads to suboptimal performance. Completely eliminating such performance degradation requires rewriting the application using synchronization primitives better than spinning. Over the benchmark set, the optimizations improve the performance of Tardis by 7.5% with respect to the baseline Tardis and 1.1% (up to 9.8%) with respect to baseline directory protocol.

Fig. 6 shows the network traffic breakdown for the same four configurations as in Fig. 5. For each experiment, we show *dram traffic*, *common traffic*, *renew traffic* and *invalidation traffic*. *Common traffic* is the traffic in common for both directory coherence

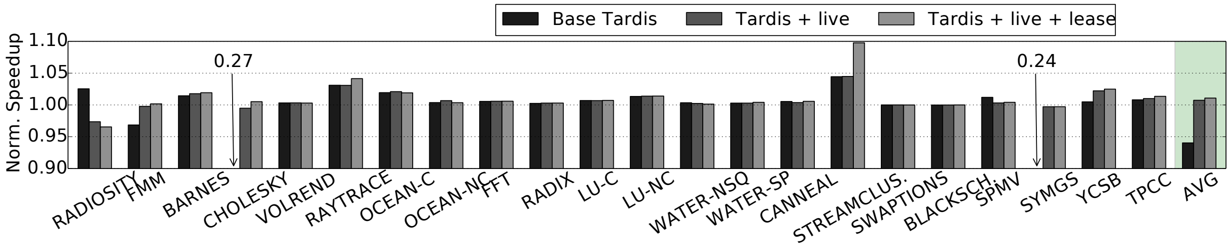


Figure 5: Speedup of livelock detection and lease prediction optimizations. All results normalized to the directory baseline.

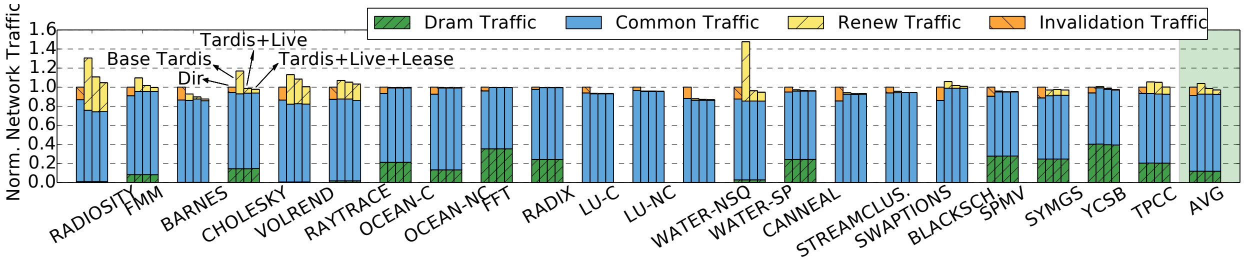


Figure 6: Network Traffic breakdown of directory and different Tardis configurations. All results normalized to the directory baseline.

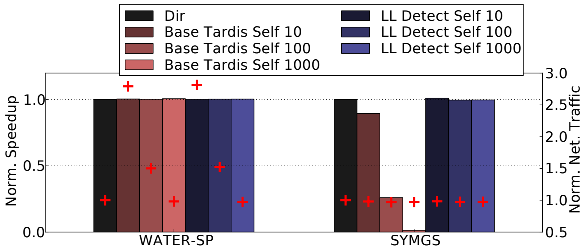


Figure 7: Sweep the *lts* self increment rate.

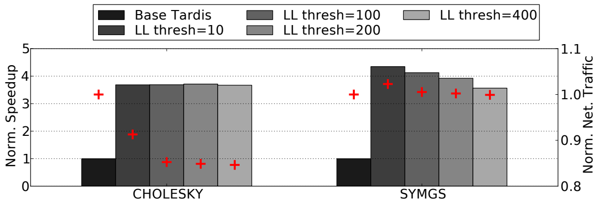


Figure 8: Sweep the livelock check threshold (*thresh_count*).

and Tardis, including shared, exclusive and write back memory requests and responses. *Renew traffic* is specific to Tardis including renew and check requests and responses. *Invalidation Traffic* is specific to the directory-based protocol, including the invalidation requests to shared copies from the directory, as well as the messages sent between L1 and LLC when a shared cacheline is evicted.

Compared to a directory protocol, Tardis is able to remove all the invalidation traffic. However, the renew traffic adds extra overhead. The baseline Tardis configuration incurs a large amount of renew traffic on some benchmarks (e.g., `RADIOSITY`, `CHOLESKY`, `VOLREND` and `WATER-SP`). Some of the renew traffic is due to fast self incrementing *lts* (e.g., `RADIOSITY`, `CHOLESKY` and `WATER-SP`). For these benchmarks, the livelock detection scheme can significantly reduce the self increment rate and therefore reduce the amount of renew traffic. On average, the livelock detection algorithm reduces the total network traffic by 5.4% compared to the baseline Tardis.

For some benchmarks, shared cachelines expire because the *lts* jumps ahead due to a write (e.g., `VOLREND`, `RADIOSITY`) and renew messages are generated. Our lease prediction algorithm is able to reduce these unnecessary renewals by using a larger lease for read intensive cachelines. On top of the livelock detection optimization, lease prediction further reduces the total network traffic by 1.5% on average. With both livelock detection and lease prediction, Tardis can reduce the total network traffic by 2.9% (up to 12.4%) compared to the baseline directory protocol.

Although not shown here, we also evaluated an idealized leasing scheme which is modeled by giving each renew message zero overhead. The idealized scheme has similar performance as the optimized Tardis but eliminates almost all the renewal messages; some renewals are still needed if the data has actually been changed.

6.3.2 Hardware Complexity

The hardware overhead for the livelock detector and lease predictor is generally small. Each livelock detector contains 8 AHB entries and each entry requires an address and a counter. Assuming 48-bit address space and a counter size of 2 bytes, the detector only requires 64 bytes of storage per core.

To implement the lease predictor, we need to store the current lease for each LLC and L1 cacheline. The lease is also transferred for each shared request and response. However, there is no need to store or transfer the whole lease value. Since a lease can only take a small number of possible values (e.g., 4 in our evaluation), we can use a smaller number of bits (e.g., 2 bits) to encode a lease, and the storage overhead is less than 0.4% in the LLC and in each L1.

6.4 Sensitivity Study

6.4.1 Self Increment Rate

Fig. 7 shows the performance and network traffic of Tardis sweeping the self increment period with and without livelock detection. All numbers are normalized to a baseline directory protocol. The `Base Tardis Self 100` is the baseline Tardis configuration and `LL Detect Self 1000` is the default optimized Tardis configuration (`LL Detect` stands for livelock detect).

In `WATER-SP`, changing the self incrementing rate does not affect performance regardless of whether livelock detection is turned on or not. This is because `WATER-SP` does not have spinning and renewals are unnecessary. Having a large self increment period significantly reduces the number of unnecessary renewals as well as the total network traffic.

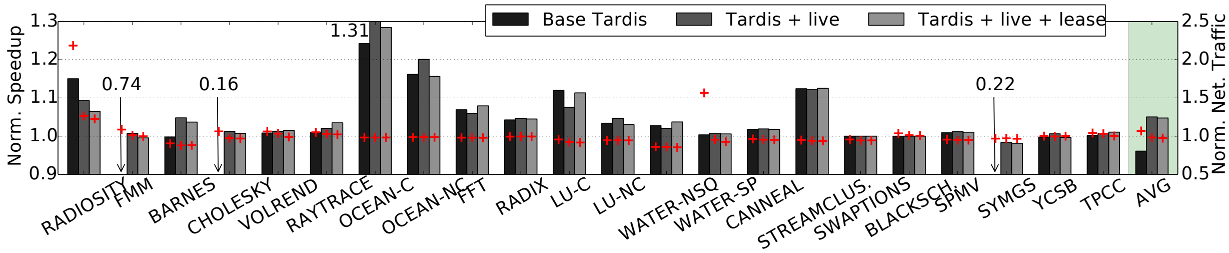


Figure 9: Tardis vs. Directory on 256 cores.

In SYMGS, for Tardis without livelock detection, performance is very sensitive to the self increment period because SYMGS intensively uses spinning to communicate between threads. If self increment is less frequent, a thread waits longer for the stale data to expire and thus performance degrades. With livelock detection, however, check requests are sent when spinning (potential livelock) is detected. Therefore, the latest value of a cacheline spun on can be returned much earlier. Regardless of the self increment period, Tardis with the livelock detector can always match the performance of the baseline directory protocol.

6.4.2 Address History Buffer Size

We swept the number of entries in the address history buffer in a livelock detector for CHOLESKY and SYMGS. According to the results (not shown), as long as the AHB buffer size is no less than 2, performance does not change. This is because in both (and most other) programs, spinning only involves a very small number of distinct memory addresses. CHOLESKY only spins on two addresses and SYMGS only spins on one address. We used a buffer size 8 by default but smaller buffers also work.

There do exist benchmarks where useful work is done during spinning and thus more than 8 distinct addresses are involved (e.g., RADIOSITY). Here, livelock detection is ineffective and correctness is guaranteed by the self incrementing program timestamp.

6.4.3 Livelock Threshold Counter

Fig. 8 shows the performance and network traffic normalized to the baseline Tardis when sweeping the minimal threshold counter (*min_counter* in Algorithm 2) in the livelock detector. The maximal threshold counter is always 8 times of the minimal value. Whenever an address in the AHB has been accessed *thresh_count* times, a check request is sent. With a larger *thresh_count*, checks are sent after spinning for a longer time which may hurt performance. On the other hand, larger *thresh_count* can reduce the total number of check messages and network traffic. In practice, the *thresh_count* should be chosen to balance the tradeoff. We chose 100 as the default threshold counter.

6.4.4 Scalability

Finally, Fig. 9 shows the performance and network traffic (normalized to baseline directory) of all benchmarks running on a 256-core system. Compared to the baseline directory, the optimized Tardis outperforms by 4.7% (upto 36.8%) and reduces the network traffic by 2.6% (upto 14.6%). Although not shown in the graph, we also evaluated Tardis and directory where both schemes consume the same area overhead. Since Tardis requires less area than directory for coherence meta data, it can have a 37% larger LLC. In area normalized evaluation, Tardis can outperform the baseline directory by 6% on average.

Note that at 256 cores, the performance improvement of Tardis is greater than the 64 core case. This indicates that Tardis not only has better scalability in terms of storage as core count increases, it also scales better in terms of performance.

7. RELATED WORK

Memory coherence is an important issue in shared memory systems with private storage in each core or processor. It has been widely studied and implemented in multicore processors [4, 5], multi-socket systems [28, 29] and distributed shared memory systems [30, 31]. Traditional directory or snoopy based coherence protocols enforce the global memory order in a consistency model using physical time order. They need an invalidation mechanism to guarantee correctness, and therefore either require non-scalable storage overhead (e.g., directory-based protocols) or broadcasting in the network (e.g., snoopy-based protocols).

Numerous previous works have tried to improve the scalability of directory coherence protocols [32, 8, 33, 9, 34, 6, 7]. Most of these works focused on better ways to organize the directory structure to improve scalability. Compared to the full-map directory protocol, these optimizations usually hurt performance and increase the design and verification complexity.

Several previous works have proposed to simplify the hardware coherence protocol for relaxed consistency models like release consistency ([35, 36, 37]) or TSO ([38]). Like Tardis, these protocols also do not require the sharer list in the directory. Unlike Tardis, they are based on the self-invalidation mechanism where all shared cachelines in an L1 cache need to be invalidated if the consistency model might be violated. For release consistency self invalidation happens for every synchronization instruction and for TSO it happens for every L1 cache miss.

Among papers along this line of research, TSO-CC [38] is the one most similar to Tardis TSO. However, since TSO-CC is self-invalidation based, it would incur more L1 misses than Tardis for benchmarks with a lot of fences. Another advantage of Tardis over TSO-CC is that Tardis is able to efficiently support any consistency model with minimal hardware reconfiguration. Running a legacy SC program on TSO-CC hardware, however, will end up with suboptimal performance and efficiency.

8. CONCLUSION

In this paper, several optimization techniques have been applied to Tardis, a very scalable physiological time based cache coherence protocol. Further, more relaxed consistency models such as the Total Store Order (TSO), Partial Store Order (PSO) and Release Consistency (RC) models are now supported in Tardis. On our set of benchmarks, evaluations indicate that optimized Tardis is better than a full-map directory protocol in terms of performance, energy and storage while being simpler.

Acknowledgement: This research was partially supported by MIT PRIMES. We thank Larry Rudolph for helping us improve this paper.

9. REFERENCES

[1] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” ACM SIGARCH Computer Architecture News, vol. 12, no. 3, pp. 348–354, 1984.

[2] 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.

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

[4] “Tile-gx family of multicore processors.” http://www.tilera.com.

[5] Intel, “Intel Xeon Phi Coprocessor System Software Developers Guide,” 2014.

[6] 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, pp. 99–110, ACM, 2010.

[7] 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, pp. 1–12, IEEE, 2012.

[8] 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.

[9] 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.

[10] X. Yu and S. Devadas, “Tardis: Time traveling coherence algorithm for distributed shared memory,” in Parallel Architectures and Compilation Techniques (PACT), 2015 International Conference on, 2015.

[11] X. Yu, M. Vijayaraghavan, and S. Devadas, “A Proof of Correctness for the Tardis Cache Coherence Protocol,” CoRR, vol. abs/1505.06459, May 2015.

[12] 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.

[13] D. L. Weaver and T. Germond, “The SPARC Architecture Manual,” 1994.

[14] D. Seal, ARM architecture reference manual. Pearson Education, 2001.

[15] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. Martin, P. Sewell, and D. Williams, “An axiomatic memory model for power multiprocessors,” in Computer Aided Verification, pp. 495–512, Springer, 2012.

[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] P. A. Bernstein, D. Shipman, and W. Wong, “Formal aspects of serializability in database concurrency control,” IEEE Transactions on Software Engineering, vol. 5, no. 3, pp. 203–216, 1979.

[18] D. P. Reed, Naming and synchronization in a decentralized computer system. PhD thesis, Massachusetts Institute of Technology, 1978.

[19] 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.

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

[21] 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, pp. 355–364, 1991.

[22] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, vol. 18. ACM, 1990.

[23] 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.

[24] 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.

[25] C. Bienia, Benchmarking Modern Multiprocessors. PhD thesis, Princeton University, January 2011.

[26] J. Dongarra, “Toward a new metric for ranking high performance computing systems,” 2013.

[27] 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.

[28] 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, pp. 1–6, IEEE, 2010.

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

[30] 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.

[31] 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.

[32] 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), pp. 353–362, ACM, 1998.

[33] 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.

[34] 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.

[35] 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, pp. 155–166, IEEE, 2011.

[36] A. Ros and S. Kaxiras, “Complexity-effective multicore coherence,” in Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pp. 241–252, ACM, 2012.

[37] H. Sung, R. Komuravelli, and S. V. Adve, “DeNovoND: Efficient hardware support for disciplined non-determinism,” ACM SIGARCH Computer Architecture News, vol. 41, no. 1, pp. 13–26, 2013.

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