TSO-CC: Consistency directed cache coherence for TSO

TSO-CC: Consistency directed cache coherence for TSO

Marco Elver
University of Edinburgh
marco.elver@ed.ac.uk

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

Abstract

Traditional directory coherence protocols are designed for the strictest consistency model, sequential consistency (SC). When they are used for chip multiprocessors (CMPs) that support relaxed memory consistency models, such protocols turn out to be unnecessarily strict. Usually this comes at the cost of scalability (in terms of per core storage), which poses a problem with increasing number of cores in today's CMPs, most of which no longer are sequentially consistent.

Because of the wide adoption of Total Store Order (TSO) and its variants in x86 and SPARC processors, and existing parallel programs written for these architectures, we propose TSO-CC, a cache coherence protocol for the TSO memory consistency model. TSO-CC does not track sharers, and instead relies on self-invalidation and detection of potential acquires using timestamps to satisfy the TSO memory consistency model lazily. Our results show that TSO-CC achieves average performance comparable to a MESI directory protocol, while TSO-CC's storage overhead per cache line scales logarithmically with increasing core count.

1. Introduction

In shared-memory chip multiprocessors (CMPs), each processor typically accesses a local cache to reduce memory latency and bandwidth. Data cached in local caches, however, can become out-of-date when they are modified by other processors. *Cache coherence* helps ensure shared memory correctness by making caches transparent to programmers.

Shared-memory correctness is defined by the *memory consistency model*, which formally specifies how the memory must appear to the programmer [1]. The relation between the processor's memory consistency model and the coherence protocol has traditionally been abstracted to the point where each subsystem considers the other as a black box [41]. Generally this is beneficial, as it reduces overall complexity; however, as a result, coherence protocols are designed for the strictest consistency model, sequential consistency (SC). SC mandates that writes are made globally visible before a subsequent memory operation. To guarantee this, before writing to a cache line, coherence protocols propagate writes *eagerly* by invalidating shared copies in other processors.

Providing eager coherence, however, comes at a cost. In snooping based protocols [2], writes to non-exclusive cache lines need to be broadcast. Scalable multiprocessors use directory based protocols [11] in which the directory maintains, for each cache line, the set of processors caching that line in the *sharing vector*. Upon a write to a non-exclusive cache line, invalidation requests are sent to only those processors caching that line. While avoiding the potentially costly broadcasts, the additional invalidation and acknowledgement messages nevertheless represents overhead [12]. More importantly, the size of the sharing vector increases linearly with the number of processors. With increasing number of processors, it could become prohibitively expensive to support a sharing vector for large-scale CMPs [12, 35].

Although there have been a number of approaches to more scalable coherence purely based on optimizing eager coherence protocols and cache organization [13, 18, 20, 29, 34, 36, 44, 48], we are interested in an alternative approach. *Lazy coherence* has generated renewed interest [7, 12, 35], as a means to address the scalability issues in eager coherence protocols. First proposed in the context of distributed shared memory (DSM) coherence [10, 22], lazy coherence protocols exploit the insight that relaxed consistency models such as *Release Consistency* (RC) require memory to be consistent only at synchronization boundaries [25]. Consequently, instead of eagerly enforcing coherence at every write, coherence is enforced lazily only at synchronization boundaries. Thus, upon a write, data is merely written to a local write-buffer, the contents of which are flushed to the shared cache upon a *release*. Upon an *acquire*, shared lines in the local caches are self-invalidated – thereby ensuring that reads to shared lines fetch the up-to-date data from the shared cache. In effect, the protocol is much simpler and *does not require a sharing vector*. To summarize, in contrast to conventional techniques for enhancing scalability, lazy coherence protocols have an added advantage since they are memory consistency directed.

1.1. Motivation

However, one important limitation of existing lazy coherence protocols concerns portability. Since they only enforce relaxed consistency models such as RC, they are not directly compatible with widely prevalent architectures such as x86 and SPARC which support variants of Total Store Order (TSO). Thus, legacy programs written for TSO could break on architectures that employ current lazy coherence protocols for RC. For this reason, it is unlikely that lazy coherence will

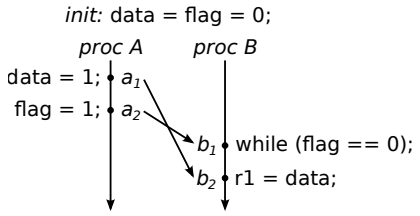

Figure 1. In TSO architectures that employ conventional (eager) coherence protocols, ① eager coherence ensures that the write a_2 to flag from $proc\ A$ becomes visible to $proc\ B$ without any additional synchronization or memory barrier. ② Once b_1 reads the value produced by a_2, TSO ordering ensures that the read b_2 sees the value written by a_1.

be adopted by the above architectures.

On the other hand, TSO coherence is amenable to a lazy coherence implementation as it relaxes the $w \rightarrow r$ ordering[1]. Writes in TSO retire into a local write-buffer and are made visible to other processors in a lazy fashion. Consequently, making them visible to other processors as soon as the writes exit the write-buffer is overkill.

1.2. Requirements

The key challenge in designing a lazy coherence protocol for TSO is the absence of explicit *release* or *acquire* instructions. Indeed, since all reads have acquire semantics and all writes have release semantics, simple writes and reads may be used for synchronization. This is illustrated in the producer-consumer example (Figure 1) in which the write a_2 is used as a release and the read b_1 is used as an acquire. In TSO, since any given write can potentially be a release, it is essential that each write is propagated to other processors, so that the value written is eventually made visible to a matching acquire – we call this the *write-propagation* requirement. In the above example, the value written by a_2 should be made visible to b_1 – or else, proc. B will keep spinning indefinitely. Additionally, TSO mandates enforcing the following three memory orderings: $w \rightarrow w$, $r \rightarrow r$, and $r \rightarrow w$ – we call this the TSO *memory ordering* requirement. In the above example, once b_1 reads the value produced by a_2, TSO ordering ensures that b_2 correctly reads the value written by a_1.

One way to trivially ensure TSO is to consider each read (write) to be an acquire (release) and naïvely use the rules of a lazy RC implementation. This, however, can cause significant performance degradation, since all reads and writes will have to be serviced by a shared cache, effectively rendering local caches useless.

1.3. Approach

In the basic scheme, for each cache line in the shared cache, we keep track of whether the line is exclusive, shared or read-only. Shared lines do *not require tracking of sharers*. Additionally, for private cache lines, we only maintain a pointer to the owner.

Since we do not track sharers, writes do not eagerly invalidate shared copies in other processors. On the contrary, writes are merely propagated to the shared cache in program order (thus ensuring $w \rightarrow w$). To save bandwidth, instead of writing the full data block to the shared cache, we merely propagate the coherence states. In the above example, the writes a_1 and a_2 are guaranteed to propagate to the shared cache in the correct order. Intuitively, the *most recent* value of any data is maintained in the shared cache.

Reads to shared cache lines are allowed to read from the local cache, up to a predefined number of accesses (potentially causing a stale value to be read), but are forced to re-request the cache line from the shared cache after exceeding an access threshold (our implementation maintains an access counter per line). This ensures that any write (used as a release) will eventually be made visible to the matching acquire, ensuring *write propagation*. In the above example, this ensures that the read b_1 will eventually access the shared cache and see the update from a_2.

When a read misses in the local cache, it is forced to obtain the most recent value from the shared cache. In order to ensure $r \rightarrow r$, future reads will also need to read the most recent values. To guarantee this, whenever a read misses in the local cache, we self-invalidate all shared cache lines. In the above example, whenever b_1 sees the update from a_2, self-invalidation ensures that b_2 correctly reads the value produced by a_1.

Finally, we reduce the number of self-invalidations by employing *timestamps* to perform transitive reduction [33]. If at a read miss, the corresponding write is determined to have happened before a previously seen write, a self-invalidation is not necessary. In the example, even though b_2 reads from the shared cache, this does not cause a self-invalidation.

1.4. Contributions

We propose TSO-CC, a coherence protocol that enforces TSO lazily without a full sharing vector. The use of a full sharing vector is an important factor that could limit scalability, which we overcome in our proposed protocol, while maintaining good overall performance in terms of execution times and on-chip network-traffic. We implemented TSO-CC in the cycle accurate full system simulator Gem5 [9], and tested our protocol implementation for adherence to TSO by running it against a set of litmus tests generated using the diy [6] tool – results indicate that TSO is indeed satisfied. TSO-CC's storage overhead per cache line scales logarithmically with increasing core count. More specifically, for 32 (128) cores, our best performing configuration reduces the storage overhead over MESI by 38% (82%). Our experiments with programs from SPLASH-2, PARSEC and STAMP benchmarks show an average reduction in execution

[1] w denotes a write, r a read and m any memory operation (write or read), $\rightarrow$ captures the happens before ordering relation between memory events.

time of 3% over the baseline (MESI), with the best case outperforming the baseline by 19%.

2. Background

In this section we first (§2.1) introduce the notion of *lazy coherence* based on definitions of relaxed memory consistency models, in particular that of Release Consistency (RC). Then we introduce the Total Store Order (TSO) memory consistency model (§2.2).

2.1. Eager versus lazy coherence

First, let us establish the relationship between the coherence protocol and consistency model. Given a target memory consistency model, the coherence protocol must ensure that memory operations become visible according to the ordering rules prescribed by the consistency model.

In SC, because all memory orderings are enforced, and in particular the $w \rightarrow r$ ordering, a write must be made visible to all processors before a subsequent read. This requirement is ensured via the use of *eager* coherence protocols which propagate writes eagerly by invalidating or updating shared cache lines in other processors [37].

On the other hand, if the consistency model is relaxed, i.e. not all possible orderings between memory operations are enforced, propagation of unordered memory operations can be delayed until an order can be re-established through synchronization boundaries [15, 25, 37]. In other words, *lazy* coherence protocols exploit the fact that relaxed consistency models require memory to be consistent only at synchronization boundaries.

A relaxed and relatively simple model which explicitly exposes synchronization operations via special instructions is RC [19]. In RC, special *release* and *acquire* instructions are used to enforce an ordering with other memory operations in program order. Given a write *release*, all memory operations prior must be visible before the write; a read *acquire* enforces preserving the program ordering of all operations after it. In addition, releases guarantee eventual propagation of synchronization data so that they become visible to corresponding acquires.

Using an eager coherence protocol in a system implementing a relaxed consistency model is potentially wasteful, as employing a lazy approach to coherence opens up further optimization opportunities to remedy the shortcomings of eager coherence protocols, as demonstrated by [4, 5, 12, 23, 25, 35, 42] in the context of RC. RC provides the optimal constraints (explicit synchronization) for a lazy coherence approach and as such is the only consistency model for which lazy coherence approaches have been studied in great detail. Typically, in a system supporting RC, lazy coherence can be implemented by ① propagating release writes and ensuring that all writes before the release are propagated first and ② upon an acquire, self-invalidating all locally cached shared data.

2.2. Total Store Order

Since TSO is the memory model found in x86 processors, its various implementations have been analyzed in detail [38]. TSO is a result of taking writes out of the critical path and entering committed writes into a FIFO write-buffer, potentially delaying a write to cache or other parts of the memory hierarchy [38, 40]. Reads to the same address as prior writes by the same processor must not be affected, which typically mandates that reads *bypass* the write-buffer.

Consequently, the write to read ordering $w \rightarrow r$ is relaxed. However, in TSO all writes have release semantics and all reads have acquire semantics. Thus, $m \rightarrow w$ and $r \rightarrow m$ need to be enforced – in other words the *TSO ordering* requirement. As every write can potentially be a release, each write needs to (eventually) propagate to other processors, so that they are made visible to a matching acquire – in other words the *write propagation* requirement.

3. Protocol Design

This section outlines the design and implementation details of the protocol: first we present a conceptual overview, followed by the basic version of the protocol, and then proceed incrementally adding optimizations to further exploit the relaxations of TSO. We assume a local L1 cache per core and a NUCA [24] architecture for the shared L2 cache.

3.1. Conceptual overview

To keep the TSO-CC protocol scalable, we do not want to use a full sharing vector. Thus, a major challenge is to enforce TSO without a full sharing vector, while minimizing costly invalidation messages – a consequence of which is that the resulting protocol must enforce coherence lazily.

Our basic approach is as follows. When a write retires from the write-buffer, instead of eagerly propagating it to all sharers like a conventional eager coherence protocol, we merely propagate the write to the shared cache. One way to do this is to simply write-though to the shared cache. To save bandwidth, however, our protocol uses a write-back policy, in that, only state changes are propagated to the shared cache. In addition to this, by delaying subsequent writes until the previous write's state changes have been acknowledged by the shared cache, we ensure that writes are propagated to the shared cache in program order. Informally, this ensures that the "most recent" value of any address can be obtained by sending a request to the shared cache.

Consequently, one way to ensure write propagation trivially is for all reads to read from the shared cache [34]. Note that this would ensure that all reads would get the most recent value, which in turn would ensure that any write which is used as a release (i.e. a synchronization operation) would definitely be seen by its matching acquire. However, the obvious problem with this approach is that it effectively means that shared data cannot be cached, which can affect performance significantly as we will show later with our experiments.

We ensure write-propagation as follows. First, let us note that ensuring write-propagation means that a write is *eventually* propagated to all processors. The keyword here is *eventually*, as there is no guarantee on *when* the propagation will occur even for shared memory systems that enforce the strongest memory consistency model (SC) using eager coherence. Consequently, shared memory systems must be programmed to work correctly even in the presence of propagation delays. While this is typically accomplished by employing proper *synchronization*, unsynchronized operations are used in shared memory systems as well. For example, synchronization constructs themselves are typically constructed using unsynchronized writes (releases) and reads (acquires). The same rules apply even with unsynchronized operations. Shared memory systems using unsynchronized operations may rely on the fact that an unsynchronized write (for e.g. release) would eventually be made visible to a read (for e.g. acquire), but must be tolerant to propagation delays. In other words, the corresponding unsynchronized read (acquire) must continually read the value to see if the write has propagated. This is precisely why all acquire-like operations have a polling read to check the synchronization value [43, 46]. This is our key observation.

Motivated by this observation, we use a simple scheme in which shared reads are allowed to hit in the local cache a predefined number of times, before forcing a miss and reading from the lower-level cache. This guarantees that those reads that are used as acquires will definitely see the value of the matching release, while ensuring that other shared data are allowed to be cached. It is important to note that in doing this optimization, we are not restricting ourself to any particular shared-memory programming model. Indeed, our experiments show that our system can work correctly for a wide variety of lock-based and lock-free programs.

Having guaranteed write-propagation we now explain how we ensure the memory orderings guaranteed by TSO. We already explained how, by propagating writes to the shared cache in program order, we ensure the $w \rightarrow w$ ordering. Ensuring the $r \rightarrow r$ ordering means that the second read should appear to perform after the first read. Whenever a read is forced to obtain its value from the shared cache (due to a miss – capacity/cold, or shared read that exceeded the maximum allowed accesses), and the last writer is not the requesting core, we *self-invalidate all shared cache lines* in the local cache. This ensures that future reads are forced to obtain the most recent data from the shared cache, thereby ensuring $r \rightarrow r$ ordering; $r \rightarrow w$ is trivially ensured as writes retire into the write-buffer only after all preceding reads complete.

3.2. Basic Protocol

Having explained the basic approach, we now discuss in detail our protocol[2]. First, we start with the basic states, and explain the actions for reads, writes, and evictions.

Stable states: The basic protocol distinguishes between invalid (Invalid), private (Exclusive, Modified) and shared (Shared) cache lines, but does not require maintaining a sharing vector. Instead, in the case of private lines – state Exclusive in the L2 – the protocol only maintains a pointer b.owner, tracking which core owns a line; shared lines are untracked in the L2. The L2 maintains an additional state Uncached denoting that no L1 has a copy of the cache line, but is valid in the L2.

Reads: Similar to a conventional MESI protocol, read requests (GetS) to invalid cache lines in the L2 result in Exclusive responses to L1s, which must acknowledge receipt of the cache line. If, however, a cache line is already in private state in the L2, and another core requests read access to the line, the request is forwarded to the owner. The owner will then downgrade its copy to the Shared state, forward the line to the requester and sends an acknowledgement to the L2, which will also transition the line to the Shared state. On subsequent read requests to a Shared line, the L2 immediately replies with Shared data responses, which do not require acknowledgement by L1s.

Unlike a conventional MESI protocol, Shared lines in the L1 are allowed to hit upon a read, *only* until some predefined maximum number of accesses, at which point the line has to be re-requested from the L2. This requires extra storage for the access counter b.acnt – the number of bits depend on the maximum number of L1 accesses to a Shared line allowed.

As Shared lines are untracked, each L1 that obtains the line must eventually self-invalidate it. *After any L1 miss, on the data response*, where the last writer is not the requesting core, *all Shared lines must be self-invalidated*.

Writes: Similar to a conventional MESI protocol, a write can only hit in the L1 cache if the corresponding cache line is held in either Exclusive or Modified state; transitions from Exclusive to Modified are silent. A write misses in the L1 in any other state, causing a write request (GetX) to be sent to the L2 cache and a wait for response from the L2. Upon receipt of the response from the L2, the local cache line's state changes to Modified and the write hits in the L1, finalizing the transition with an acknowledgement to the L2. The L2 cache must reflect the cache line's state with the Exclusive state and set b.owner to the requester's id. If another core requests write access to a private line, the L2 sends an invalidation message to the owner stored in b.owner, which will then pass ownership to the core which requested write access. Since the L2 only responds to write requests if it is in a stable state, i.e. it has received the acknowledgement of the last writer, there can only be one writer at a time. This serializes all writes to the same address at the L2 cache.

Unlike a conventional MESI protocol, on a write to a Shared line, the L2 immediately responds with a data response message and transitions the line to Exclusive. Note that even if the cache line is in Shared, the L2 must send the

[2] A detailed state transition table is available online: http://homepages.inf.ed.ac.uk/s0787712/research/tsocc

entire line, as the requesting core may have a stale copy. On receiving the data message, the L1 transitions to Exclusive either from Invalid or Shared. Note that there may still be other copies of the line in Shared in other L1 caches, but since they will eventually re-request the line and subsequently self-invalidate all Shared lines, TSO is satisfied.

Evictions: Cache lines which are untracked in the L2 do not need to be inclusive. Therefore, on evictions from the L2, only Exclusive line evictions require invalidation requests to the owner; Shared lines are evicted from the L2 silently. Similarly for the L1, Exclusive lines need to inform the L2, which can then transition the line to Uncached; Shared lines are evicted silently.

3.3. Opt. 1: reducing self-invalidations

In order to satisfy the $r \rightarrow r$ ordering, in the basic protocol, all L2 accesses except to lines where b.owner is the requester, result in self-invalidation of all Shared lines. This leads to shared accesses following an acquire to miss and request the cache line from the L2, and subsequently self-invalidating all shared lines again. For example in Figure 1, self-invalidating all Shared lines on the acquire b_1 but also on subsequent read misses is not required. This is because, the self-invalidation at b_1 is supposed to make all writes before a_2 visible. Another self-invalidation happens at b_2 to make all writes before a_1 visible. However, this is unnecessary, as the self-invalidation at b_1 (to make all writes before a_2 visible) has already taken care of this. To reduce unnecessary invalidations, we implement a version of the transitive reduction technique outlined in [33].

Each line in the L2 and L1 must be able to store a timestamp b.ts of fixed size; the size of the timestamp depends on the storage requirements, but also affects the frequency of timestamp resets, which are discussed in more detail in §3.5. A line's timestamp is updated on every write, and the source of the timestamp is a unique, monotonically increasing core local counter, which is incremented on every write.

Thus, to reduce invalidations, only where the requested line's timestamp is *larger than the last-seen timestamp from the writer of that line*, treat the event as a *potential acquire* and self-invalidate all Shared lines.

To maintain the list of last-seen timestamps, each core maintains a timestamp table ts_L1. The maximum possible entries per timestamp table can be less than the total number of cores, but will require an eviction policy to deal with limited capacity. The L2 responds to requests with the data, the writer b.owner and the timestamp b.ts. For those data responses where the timestamp is invalid (lines which have never been written to since the L2 obtained a copy) or there does not exist an entry in the L1's timestamp-table (never read from the writer before), it is also required to self-invalidate; this is because timestamps are not propagated to main-memory and it may be possible for the line to have been modified and then evicted from the L2.

Timestamp groups: To reduce the number of timestamp resets, it is possible to assign groups of contiguous writes the same timestamp, and increment the local timestamp-source after the maximum writes to be grouped is reached. To still maintain correctness under TSO, this changes the rule for when self-invalidation is to be performed: only where the requested line's timestamp is *larger or equal* (contrary to just larger as before) *than the last-seen timestamp from the writer of that line*, self-invalidate all Shared lines.

3.4. Opt. 2: shared read-only data

The basic protocol does not take into account lines which are written to very infrequently but read frequently. Another problem are lines which have no valid timestamp (due to prior L2 eviction), causing frequent mandatory self-invalidations. To resolve these issues, we add another state SharedRO for shared read-only cache lines.

A line transitions to SharedRO instead of Shared if the line is not modified by the previous Exclusive owner (this prevents Shared lines with invalid timestamps). In addition, cache lines in the Shared state *decay* after some predefined time of not being modified, causing them to transition to SharedRO. In our implementation, we compare the difference between the shared cache line's timestamp and the writer's last-seen timestamp maintained in a table of last-seen timestamps ts_L1 in the L2 (this table is reused in §3.5 to deal with timestamp resets). If the difference between the line's timestamp and last-seen timestamp exceeds a predefined value, the cache line is transitioned to SharedRO.

Since on a self-invalidation, only Shared lines are invalidated, this optimization already decreases the number of self-invalidations, as SharedRO lines are excluded from invalidations. Regardless, this still poses an issue, as on every SharedRO data response, the timestamp is still invalid and will cause self-invalidations. To solve this, we introduce timestamps for SharedRO lines with the timestamp-source being the L2 itself – note that each L2 tile will maintain its own timestamp-source. The event on which a line is assigned a timestamp is on transitions from Exclusive or Shared to SharedRO. On such transitions the L2 tile increments its timestamp-source.

Each L1 must maintain a table ts_L2 of last-seen timestamps for each L2 tile. On receiving a SharedRO data line from the L2, the following rule determines if self-invalidation should occur: if the line's timestamp is *larger than the last-seen timestamp from the L2*, self-invalidate all Shared lines.

Writes to shared read-only lines: A write request to a SharedRO line requires a broadcast to all L1s to invalidate the line. To reduce the number of required broadcast invalidation and acknowledgement messages, the b.owner entry in the L2 directory is reused as a coarse sharing vector [20], where each bit represents a group of sharers; this permits SharedRO evictions from L1 to be silent. As writes to SharedRO lines

should be infrequent, the impact of unnecessary SharedRO invalidation/acknowledgement messages should be small.

Timestamp groups: To reduce the number of timestamp resets, the same timestamp can be assigned to groups of SharedRO lines. In order to maintain $r \rightarrow r$ ordering, a core must self-invalidate on a read to a SharedRO line that could potentially have been modified since the last time it read the same line. This can only be the case, if a line ends up in a state, after a modification, from which it can reach SharedRO again: ① only after a L2 eviction of a dirty line; after a GetS request to a line in Uncached which has been modified; or ② after a line enters the Shared state. It suffices to have a flag for conditions ① and ② each to denote if the timestamp-source should be incremented on a transition event to SharedRO. All flags are reset after incrementing the timestamp-source.

3.5. Timestamp resets

Since timestamps are finite, we have to deal with timestamp resets for both L1 and L2 timestamps. If the timestamp and timestamp-group size are chosen appropriately, timestamp resets should occur relatively infrequently, and does not contribute overly negative to network traffic. As such, the protocol deals with timestamp resets by requiring the node, be it L1 or L2 tile, which has to reset its timestamp-source to broadcast a timestamp reset message.

In the case where a L1 requires resetting the timestamp-source, the broadcast is sent to every other L1 and L2 tile. Upon receiving a timestamp reset message, a L1 invalidates the sender's entry in the timestamp table ts_L1. However, it is possible to have lines in the L2 where the timestamp is from a previous epoch, where each epoch is the period between timestamp resets, i.e. b.ts is larger than the current timestamp-source of the corresponding owner. The only requirement is that the L2 must respond with a timestamp that reflects the correct happens-before relation.

The solution is for each L2 tile to maintain a table of last-seen timestamps ts_L1 for every L1; the corresponding entry for a writer is updated when the L2 updates a line's timestamp upon receiving a data message. Every L2 tile's last-seen timestamp table must be able to hold as many entries as there are L1s. The L2 will assign a data response message the line's timestamp b.ts if the *last-seen timestamp from the owner is larger or equal to b.ts, the smallest valid timestamp otherwise*. Similarly for requests forwarded to a L1, only that the line's timestamp is compared against the current timestamp-source.

Upon resetting a L2 tile's timestamp, a broadcast is sent to every L1. The L1s remove the entry in ts_L2 for the sending tile. To avoid sending larger timestamps than the current timestamp-source, the same rule as for responding to lines not in SharedRO as described in the previous paragraph is applied (compare against L2 tile's current timestamp-source).

One additional case must be dealt with, such that if the smallest valid timestamp is used if a line's timestamp is from a previous epoch, it is not possible for a L1 to skip self-invalidation due to the line's timestamp being equal to the smallest valid timestamp. To address this case, the next timestamp assigned to a line after a reset must always be larger than the smallest valid timestamp.

Handling races: As it is possible for timestamp reset messages to race with data request and response messages, the case where a data response with a timestamp from a previous epoch arrives at a L1 which already received a timestamp reset message, needs to be accounted for. Waiting for acknowledgements from all nodes having a potential entry of the resetter in a timestamp table would cause twice the network traffic on a timestamp reset and unnecessarily complicates the protocol. We introduce an *epoch-id* to be maintained per timestamp-source. The epoch-id is incremented on every timestamp reset and the new epoch-id is sent along with the timestamp reset message. It is not a problem if the epoch-id overflows, as the only requirement for the epoch-id is to be distinct from its previous value. However, we assume a bound on the time it takes for a message to be delivered, and it is not possible for the epoch-id to overflow and reach the same epoch-id value of a message in transit.

Each L1 and L2 tile maintains a table of epoch-ids for every other node: L1s maintain epoch-ids for every other L1 (epoch_ids_L1) and L2 (epoch_ids_L2) tile; L2 tiles maintain epoch-ids for all L1s. Every data message that contains a timestamp, must now also contain the epoch-id of the source of the timestamp: the owner's epoch-id for non-SharedRO lines and the L2 tile's epoch-id for SharedRO lines.

Upon receipt of a data message, the L1 compares the expected epoch-id with the data message's epoch-id; if they do not match, the same action as on a timestamp reset has to be performed, and can proceed as usual if they match.

3.6. Atomic accesses and fences

Implementing atomic read and write instructions, such as RMWs, is trivial with our proposed protocol. Similarly to MESI protocols, in our protocol an atomic instruction also issues a GetX request. Fences require unconditional self-invalidation of cache lines in the Shared state.

3.7. Storage requirements & organization

Table 1 shows a detailed breakdown of storage requirements for a TSO-CC implementation, referring to literals introduced in §3. Per cache line storage requirements has the most significant impact, which scales logarithmically with increasing number of cores (see §4, Figure 2).

While we chose a simple sparse directory embedded in the L2 cache for our evaluation (Figure 2), our protocol is independent of a particular directory organization. It is possible to further optimize our overall scheme by using directory organization approaches such as in [18, 36]; however, this is beyond the scope of this paper. Also note that we do not require inclusivity for Shared lines, alleviating some of the set conflict issues associated with the chosen organization.

Table 1. TSO-CC specific storage requirements.

L1	**Per node:** • Current timestamp, B_{ts} bits • Write-group counter, $B_{write-group}$ bits • Current epoch-id, $B_{epoch-id}$ bits • Timestamp-table ts_L1$[n]$, $n \leq Count_{L1}$ entries • Epoch-ids epoch_ids_L1$[n]$, $n = Count_{L1}$ entries Only required if SharedRO opt. (§3.4) is used: • Timestamp-table ts_L2$[n]$, $n \leq Count_{L2-tiles}$ entries • Epoch-ids epoch_ids_L2$[n]$, $n = Count_{L2-tiles}$ entries **Per line b:** • Number of accesses b.acnt, B_{maxacc} bits • Last-written timestamp b.ts, B_{ts} bits
L2	**Per tile:** • Last-seen timestamp-table ts_L1, $n = Count_{L1}$ entries • Epoch-ids epoch_ids_L1$[n]$, $n = Count_{L1}$ entries Only required if SharedRO opt. (§3.4) is used: • Current timestamp, B_{ts} bits • Current epoch-id, $B_{epoch-id}$ bits • Increment-timestamp-flags, 2 bits **Per line b:** • Timestamp b.ts, B_{ts} bits • Owner (Exclusive), last-writer (Shared), sharer-count (SharedRO) as b.owner, $log(Count_{L1})$ bits

4. Evaluation Methodology

This section provides an overview of our evaluation methodology used in obtaining the performance results (§5).

4.1. Simulation Environment

Simulator: For the evaluation of TSO-CC, we use the Gem5 simulator [9] in Ruby *full-system* mode. GARNET [3] is used to model the on-chip interconnect. The ISA used is x86_64, as it is the most widely used architecture that assumes a variant of TSO. The processor model used for each CMP core is a simple out-of-order processor. Table 2 shows the key-parameters of the system.

As TSO-CC explicitly allows accesses to stale data, this needs to be reflected in the *functional execution* (not just the timing) of the simulated execution traces. We added support to the simulator to functionally reflect cache hits to stale data, as the stock version of Gem5 in full-system mode would assume the caches to always be coherent otherwise.

Workloads: Table 3 shows the benchmarks we have selected from the PARSEC [8], SPLASH-2 [45] and STAMP [31] benchmarks suites. The STAMP benchmark suite has been chosen to evaluate transactional synchronization compared to the more traditional approach from PARSEC and SPLASH-2; the STM algorithm used is *NOrec* [14].

Note that in the evaluated results, we include two versions of *lu*, with and without the use of contiguous block allocation. The version which makes use of contiguous block allocation avoids false sharing, whereas the non-contiguous version does not. Both version are included to show the effect of false-sharing, as previous works have shown lazy protocols to perform better in the presence of false-sharing [16].

All selected workloads correctly run to completion with both MESI and our configurations. It should also be emphasized that all presented program codes run unmodified (including the Linux kernel) with the TSO-CC protocol.

Table 2. System parameters.

Core-count & frequency	32 (out-of-order) @ 2GHz
Write buffer entries ROB entries	32, FIFO 40
L1 I+D -cache (private) L1 hit latency	32KB+32KB, 64B lines, 4-way 3 cycles
L2 cache (NUCA, shared) L2 hit latency	1MB×32 tiles, 64B lines, 16-way 30 to 80 cycles
Memory Memory hit latency	2GB 120 to 230 cycles
On-chip network	2D Mesh, 4 rows, 16B flits
Kernel	Linux 2.6.32.60

Table 3. Benchmarks and their input parameters.

PARSEC	blackscholes	simmedium
	canneal	simsmall
	dedup	simsmall
	fluidanimate	simsmall
	x264	simsmall
SPLASH-2	fft	64K points
	lu	512×512 matrix, 16×16 blocks
	radix	256K, radix 1024
	raytrace	car
	water-nsquared	512 molecules
STAMP	bayes	-v32 -r1024 -n2 -p20 -i2 -e2
	genome	-g512 -s32 -n32768
	intruder	-a10 -l4 -n2048 -s1
	ssca2	-s13 -i1.0 -u1.0 -l3 -p3
	vacation	-n4 -q60 -u90 -r16384 -t4096

4.2. Parameters & storage overheads

In order to evaluate our claims, we have chosen the MESI directory protocol implementation in Gem5 as the baseline. To assess the performance of TSO-CC, we have selected a range of configurations to show the impact of varying the timestamp and write-group size parameters.

We start out with a basic selection of parameters which we derived from a limited design-space exploration. We have determined 4 bits for the per-line access counter to be a good balance between average performance and storage-requirements, since higher values do not yield a consistent improvement in performance; this allows at most 16 consecutive L1 hits for Shared lines.

Furthermore, in all cases the shared read-only optimization as described in §3.4 contributes a significant improvement: average execution time is reduced by more than 35% and average on-chip network traffic by more than 75%. Therefore, we only consider configurations with the shared read-only optimization. The decay time (for transitioning Shared to SharedRO) is set to a fixed number of writes, as

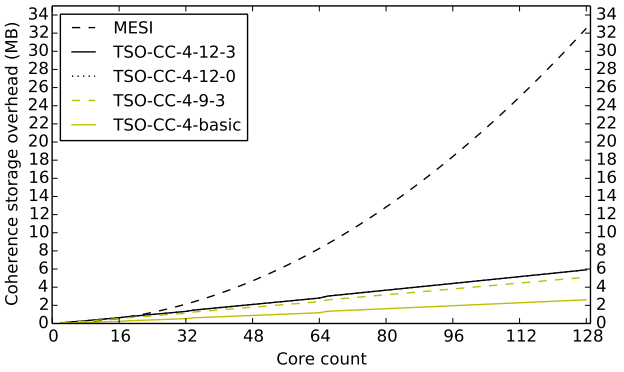

Figure 2. Storage overhead with all optimizations enabled, 1MB per L2 tile, and as many tiles as cores; the timestamp-table sizes match the number of cores and L2 tiles; $B_{epoch-id} = 3$ bits per epoch-id.

reflected by the timestamp (taking into account write-group size); we have determined 256 writes to be a good value.

Below we consider the following configurations: CC-shared-to-L2, TSO-CC-4-basic, TSO-CC-4-noreset, TSO-CC-4-12-3, TSO-CC-4-12-0, TSO-CC-4-9-3. From the parameter names introduced in Table 1, the naming convention used is TSO-CC-B_{maxacc}-B_{ts}-$B_{write-group}$.

CC-shared-to-L2: A simple protocol that removes the sharing list, but as a result, reads to Shared lines always miss in the L1 and must request the data from the L2. The base protocol implementation is the same as TSO-CC, and also includes the shared read-only optimization (without the ability to decay Shared lines, due to no timestamps). With a system configuration as in Table 2, CC-shared-to-L2 reduces coherence storage requirements by 76% compared to MESI.

TSO-CC-4-basic: An implementation of the protocol as described in §3.2 with the shared read-only optimization. Over CC-shared-to-L2, TSO-CC-4-basic only requires additional storage for the per L1 line accesses counter. TSO-CC-4-basic reduces storage requirements by 75% for 32 cores.

TSO-CC-4-noreset: Adds the optimization described in §3.3, but assumes infinite timestamps[3] to eliminate timestamp reset events, and increments the timestamp-source on every write, i.e. write-group size of 1. This configuration is expected to result in the lowest self-invalidation count, as timestamp-resets also affect invalidations negatively.

To assess the effect of the timestamp and the write-group sizes using realistic (feasible to implement) storage requirements, the following configurations have been selected.

TSO-CC-4-12-3: From evaluating a range of realistic protocols, this particular configuration results in the best trade-off between *storage*, and performance (in terms of *execution times* and *network traffic*). In this configuration 12 bits are used for timestamps and the write-group size is 8 (3 bits extra storage required per L1). The storage reduction over MESI is 38% for 32 cores.

TSO-CC-4-12-0: In this configuration the write-group size is decreased to 1, to show the effect of varying the write-group size. The reduction in storage overhead over MESI is 38% for 32 cores.

TSO-CC-4-9-3: This configuration was chosen to show the effect of varying the timestamp bits, while keeping the write-group size the same. The timestamp size is reduced to 9 bits, and write-group size is kept at 8. On-chip coherence storage overhead is reduced by 47% over MESI for 32 cores. Note that timestamps reset after the same number of writes as TSO-CC-4-12-0, but 8 times as often as TSO-CC-4-12-3.

Figure 2 shows a comparison of the extra coherence storage requirements between MESI, TSO-CC-4-12-3, TSO-CC-4-12-0, TSO-CC-4-9-3 and TSO-CC-4-basic for core counts up to 128. The best case realistic configuration TSO-CC-4-12-3 reduces on-chip storage requirements by 82% over MESI at 128 cores.

4.3. Verification

To check the protocol implementation for adherence to the consistency model, a set of litmus tests were chosen to be run in the full-system simulator. The diy [6] tool was used to generate litmus tests for TSO according to [38]. This was invaluable in finding some of the more subtle issues in the implementation of the protocol. *According to the litmus tests, each configuration of the protocol satisfies TSO.* Keeping in mind that the litmus tests are not exhaustive, we can conclude with a high level of confidence that the consistency model in the implementation is satisfied. In addition, we model checked the protocol for race conditions and deadlocks.

5. Experimental Results

This section highlights the simulation results, and additionally gives insight into how execution times and network traffic are affected by some of the secondary properties (timestamp-resets, self-invalidations).

In the following we compare the performance of TSO-CC with MESI. Figure 3 shows normalized (w.r.t. MESI) execution times and Figure 4 shows normalized network traffic (total flits) for all chosen benchmarks and configurations. For all TSO-CC configurations, we determine additional network traffic due to SharedRO-invalidations to be insignificant compared to all other traffic, as writes to SharedRO are too infrequent to be accounted for in Figure 5.

CC-shared-to-L2: We begin with showing how the naïve implementation without a sharing vector performs. On average, CC-shared-to-L2 has a slowdown of 14% over MESI; the best case, *fft*, performs 14% faster than the baseline, while the worst case has a slowdown of 84% for *lu (cont.)*. Network traffic is more sensitive, with an average increase of 137%. CC-shared-to-L2 performs poorly in cases with frequent shared misses, as seen in Figure 5, but much better in

[3]The simulator implementation uses 31 bit timestamps, which is more than sufficient to eliminate timestamp reset events for the chosen workloads.

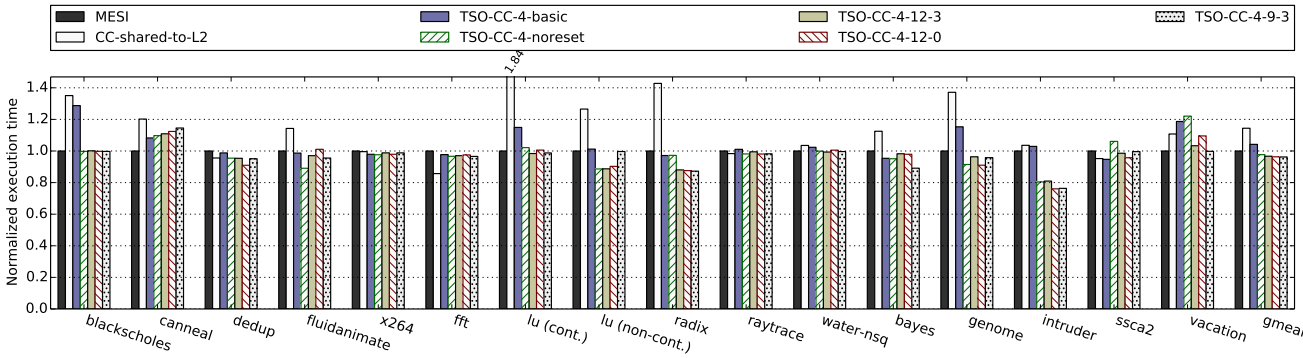

Figure 3. Execution times, normalized against MESI.

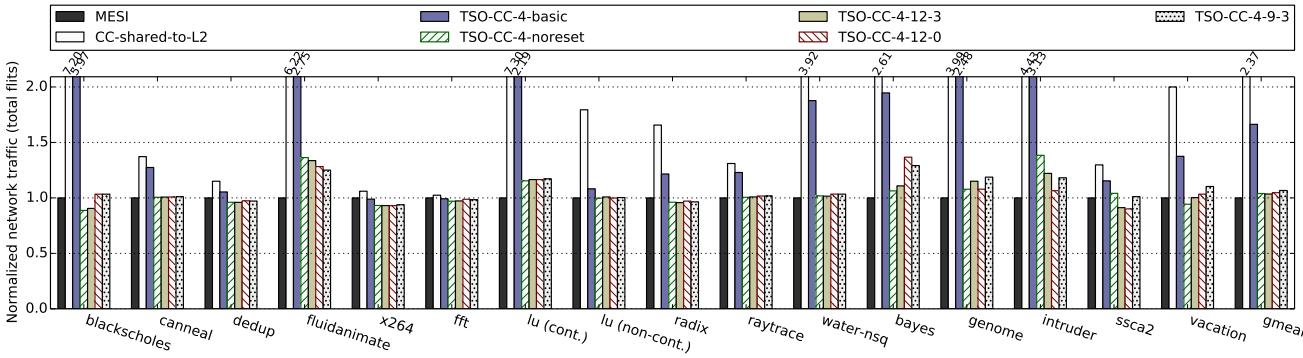

Figure 4. Network traffic (total flits), normalized against MESI.

cases with a majority of private accesses and most shared reads are to shared read-only lines, as Figure 6 shows,

TSO-CC-4-basic: Compared to the baseline, TSO-CC-4-basic is 4% slower; the patterns observed are similar to CC-shared-to-L2. The best case speedup is 5% for *ssca2*, and worst case slowdown is 29% for *blackscholes*. Allowing read hits to Shared lines until the next L2 access improves execution time compared to CC-shared-to-L2 by 9%, and network traffic by 30% on average. Since the transitive reduction optimization is not used, most L1 misses cause self-invalidation as confirmed by Figure 7; on average 40% of read misses cause self-invalidation.

TSO-CC-4-noreset: The ideal case TSO-CC-4-noreset shows an average of 2% improvement in execution time over the baseline; best case speedup of 20% for *intruder*, worst case slowdown of 22% for *vacation*. On average, self-invalidations – potential acquires detected as seen in Figure 7 – are reduced by 87%, directly resulting in a speedup of 6% over TSO-CC-4-basic. Overall, TSO-CC-4-noreset requires 4% more on-chip network traffic compared to the baseline, an improvement of 37% over TSO-CC-4-basic.

TSO-CC-4-12-3: The overall best realistic configuration is on average 3% faster than the MESI baseline. The best case speedup is 19% for *intruder*, and worst case slowdown is 10% for *canneal*. This configuration performs as well as TSO-CC-4-noreset (the ideal case), despite the fact that self-invalidations have increased by 25%. Over TSO-CC-4-basic, average execution time improves by 7%, as a result of reducing self-invalidations by 84%. The average network-traffic from TSO-CC-4-noreset (no timestamp resets) to TSO-CC-4-12-3 does not increase, which indicates that timestamp-reset broadcasts are insignificant compared to all other on-chip network traffic.

There are two primary reasons as to why TSO-CC-4-12-3 outperforms MESI. First, our protocol has the added benefit of reduced negative effects from false sharing, as has been shown to hold for lazy coherence protocols in general [16]. This is because shared lines are not invalidated upon another core requesting write access, and reads can continue to hit in the L1 until self-invalidated. This can be observed when comparing the two versions of *lu*. The version which does not eliminate false-sharing *(non-cont.)* performs significantly better with TSO-CC-4-12-3 compared to MESI, whereas the version where the programmer explicitly eliminates false-sharing *(cont.)* results in similar execution times.

Second, our protocol performs better for GetX requests (writes, RMWs) to shared cache lines, as we do not require invalidation messages to be sent to each sharer, which must also be acknowledged. This can be seen in the case of *radix*, which has a relatively high write miss rate as seen in Figure 5. Further evidence for this can be seen in Figure 8, which shows the normalized average latencies of RMWs.

As we have seen, the introduction of the transitive reduction optimization (§3.3) contributes a large improvement

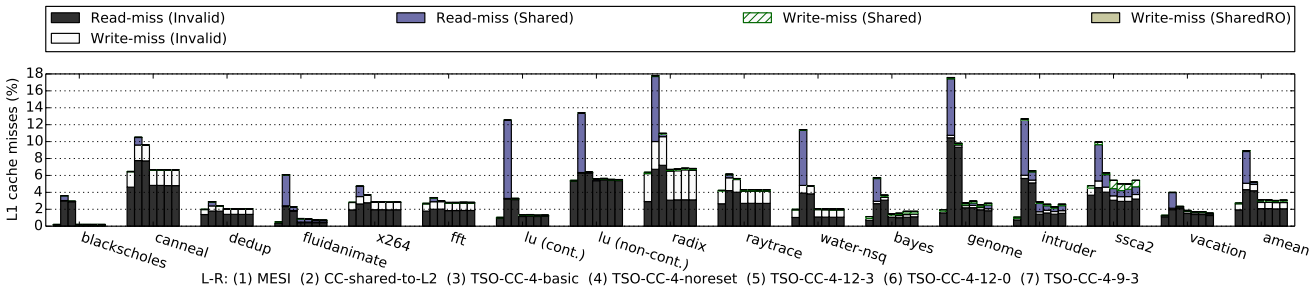

Figure 5. Detailed breakdown of L1 cache misses by Invalid, Shared and SharedRO states.

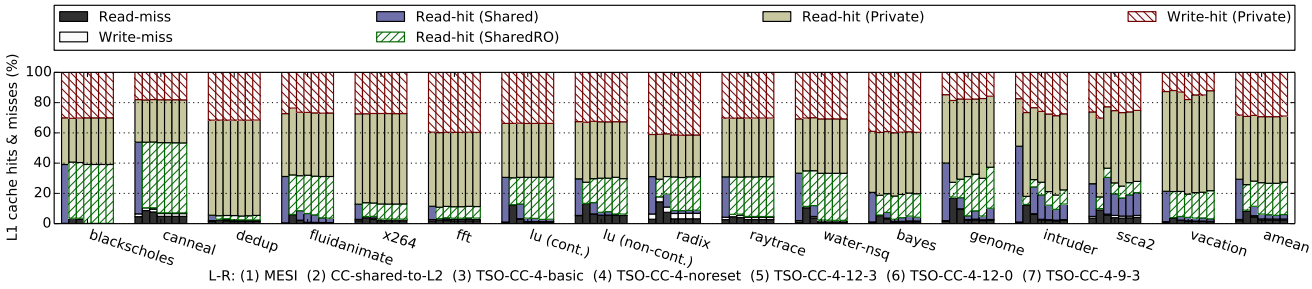

Figure 6. L1 cache hits and misses; hits split up by Shared, SharedRO and private (Exclusive, Modified) states.

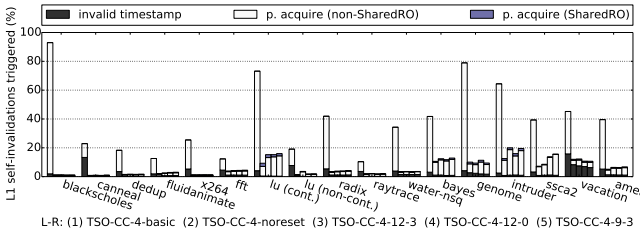

Figure 7. Percentage of L1 self-invalidation events triggered by data response messages.

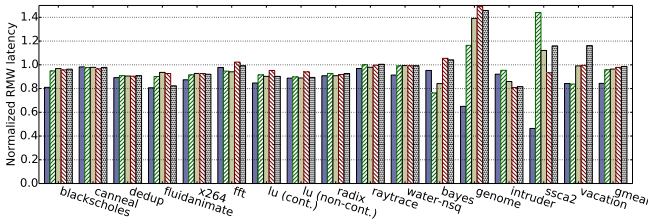

Figure 8. RMW latencies, normalized against MESI. (Legend same as Figure 3, TSO-CC only)

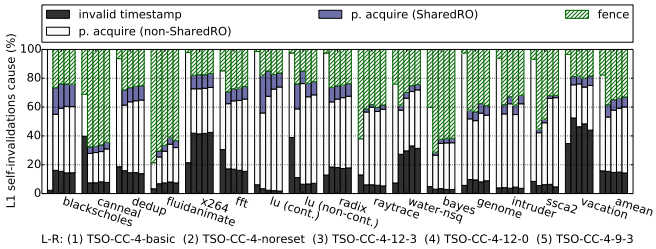

Figure 9. Breakdown of L1 self-invalidation cause.

over TSO-CC-4-basic, and next we look at how varying the TSO-CC parameters can affect performance.

TSO-CC-4-12-0: Decreasing the write-group size by a factor of 8× (compared to TSO-CC-4-12-3) results in a proportional increase in timestamp-resets, yet potential acquires detected are similar to TSO-CC-4-12-3 (Figure 7). One reason for this is that a write-group size of 1 results in more accurate detection of potential acquires, reducing self-invalidations. Thus, average execution time is similar to TSO-CC-4-12-3. However, network traffic is more sensitive; TSO-CC-4-12-0 requires 5% more network traffic compared to the baseline.

TSO-CC-4-9-3: Decreasing the maximum timestamp size by 3 bits while keeping the write-group size the same, compared to TSO-CC-4-12-3, results in an expected increase of timestamp-resets of 8×, and stays the same compared to TSO-CC-4-12-0. Because of this, and because write-groups are more coarse grained, this parameter selection results in an increase of self-invalidations by 5% (7%), yet no slowdown compared to TSO-CC-4-12-3 (TSO-CC-4-12-0). The best case is *intruder* with an improvement of 24% over MESI, the worst case is *canneal* with a slowdown of 15%. TSO-CC-4-9-3 requires 7% more network traffic compared to MESI, indicating that network traffic is indeed more sensitive to increased self-invalidations.

As both timestamp-bits and write-group size change, the number of timestamp-resets in the system change proportionally. As timestamp-resets increase, invalidation of entries in timestamp-tables increases, and as a result, upon reading a cache line where there does not exist an entry in the timestamp-table for the line's last writer, a potential acquire is forced and all Shared lines are invalidated. This trend can be observed in Figure 7. The breakdown of self-invalidation causes can be seen in Figure 9.

6. Related Work

Closely related work is mentioned in previous sections, whereas this section provides a broader overview of more scalable approaches to coherence.

Coherence for sequential consistency: Among the approaches *with a sharing vector*, are hierarchical directories [29, 44], which solve some of the storage concerns, but increase overall organization complexity through additional levels of indirection.

Coarse sharing vector approaches [20, 48] reduce the sharing vector size, however, with increasing number of cores, using such approaches for all data becomes prohibitive due to the negative effect of unnecessary invalidation and acknowledgement messages on performance. More recently, SCD [36] solves many of the storage concerns of full sharing vectors by using variable-size sharing vector representations, again with increased directory organization complexity.

Furthermore, several schemes optimize standalone sparse directory utilization [18, 36] by reducing set conflict issues. This allows for smaller directories even as the number of cores increase. Note that these approaches are orthogonal to our approach, as they optimize directory organization but not the protocol, and thus do not consider the consistency model.

Works *eliminating sharing vectors* [13, 34], observe most cache lines to be private, for which maintaining coherence is unnecessary. For example, shared data can be mapped onto shared and private data onto local caches [34], eliminating sharer tracking. However, it is possible to degrade performance for infrequently written but frequently read lines, suggested by our implementation of CC-shared-to-L2.

Coherence for relaxed consistency: Dubois and Scheurich [15, 37] first gave insight into reducing coherence overhead in relaxed consistency models, particularly that the requirement of "coherence on synchronization points" is sufficient. Instead of enforcing coherence at every write (also referred as the SWMR property [41]), recent works [7, 12, 17, 21, 28, 35, 42] enforce coherence at synchronization boundaries by self-invalidating shared data in private caches.

Dynamic Self-Invalidation (DSI) [27] proposes self-invalidating cache lines obtained as tear-off copies, instead of waiting for invalidation from directory to reduce coherence traffic. The best heuristic for self-invalidation triggers are synchronization boundaries. More recently, SARC [21] improves upon these concepts by predicting writers to limit accesses to the directory. Both [21, 27] improve performance by reducing coherence requests, but still rely on an eager protocol for cache lines not sent to sharers as tear-off copies.

Several recent proposals *eliminate sharing vector* overheads by targeting relaxed consistency models; they do not, however, consider consistency models stricter than RC. DeNovo [12], and more recently DeNovoND [42], argue that more disciplined programming models must be used to achieve less complex and more scalable hardware. DeNovo proposes a coherence protocol for data-race-free (DRF) programs, however, requires explicit programmer information about which regions in memory need to be self-invalidated at synchronization points. The work by [35], while not requiring explicit programmer information about which data is shared nor a directory with a sharing vector, present a protocol limiting the number of self-invalidations by distinguishing between private and shared data using the TLB.

Several works [30, 47] also make use of timestamps to limit invalidations by detecting the validity of cache lines based on timestamps, but require software support. Contrary to these schemes, and how we use timestamps to detect ordering, the hardware-only approaches proposed by [32, 39] use globally synchronized timestamps to enforce ordering based on predicted lifetimes of cache lines.

For distributed shared memory (DSM): The observation of only enforcing coherent memory in logical time [26] (causally), allows for further optimizations. This is akin to the relationship between coherence and consistency given in §2.1. Causal Memory [4, 5] as well as [23] make use of this observation in coherence protocols for DSM. Lazy Release Consistency [23] uses vector clocks to establish a partial order between memory operations to only enforce completion of operations which happened-before acquires.

7. Conclusion

We have presented TSO-CC, a lazy approach to coherence for TSO. Our goal was to design a more scalable protocol, especially in terms of on-chip storage requirements, compared to conventional MESI directory protocols. Our approach is based on the observation that using eager coherence protocols in the context of systems with more relaxed consistency models is unnecessary, and the coherence protocol can be optimized for the target consistency model. This brings with it a new set of challenges, and in the words of Sorin et al. [41] "incurs considerable intellectual and verification complexity, bringing to mind the Greek myth about Pandora's box".

The complexity of the resulting coherence protocol obviously depends on the consistency model. While we aimed at designing a protocol that is simpler than MESI, to achieve good performance for TSO, we had to sacrifice simplicity. Indeed, TSO-CC requires approximately as many combined stable and transient states as a MESI implementation.

Aside from that, we have constructed a more scalable coherence protocol for TSO, which is able to run unmodified legacy codes. Preliminary verification results based on litmus tests give us a high level of confidence in its correctness (further verification reserved for future work). More importantly, TSO-CC has a significant reduction in coherence storage overhead, as well as an overall reduction in execution time. Despite some of the complexity issues, we believe these are positive results, which encourages a second look at consistency-directed coherence design for TSO-like architectures. In addition to this, it would be very interesting to see

if the insights from our work can be used in conjunction with other conventional approaches for achieving scalability.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments and advice. This work is supported by the Centre for Numerical Algorithms and Intelligent Software, funded by EPSRC grant EP/G036136/1 and the Scottish Funding Council to the University of Edinburgh.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12), 1996.

[2] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. 1988.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *ISPASS*, 2009.

[4] M. Ahamad, P. W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *ICDCS*, 1991.

[5] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1), 1995.

[6] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *TACAS*, 2011.

[7] T. J. Ashby, P. Diaz, and M. Cintra. Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters. *IEEE Trans. Computers*, 60(4), 2011.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*, 2008.

[9] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2), 2011.

[10] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *SOSP*, 1991.

[11] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Trans. Computers*, 27(12), 1978.

[12] 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 *PACT*, 2011.

[13] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *ISCA*, 2011.

[14] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPOPP*, 2010.

[15] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *ISCA*, 1986.

[16] M. Dubois, J.-C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *SC*, 1991.

[17] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *HPCA*, 2008.

[18] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *HPCA*, 2011.

[19] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*, 1990.

[20] A. Gupta, W.-D. Weber, and T. C. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *ICPP (1)*, 1990.

[21] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5), 2010.

[22] P. J. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter*, 1994.

[23] P. J. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, 1992.

[24] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS*, 2002.

[25] L. I. Kontothanassis, M. L. Scott, and R. Bianchini. Lazy Release Consistency for Hardware-Coherent Multiprocessors. In *SC*, 1995.

[26] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), 1978.

[27] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *ISCA*, 1995.

[28] D. Liu, Y. Chen, Q. Guo, T. Chen, L. Li, Q. Dong, and W. Hu. DLS: Directoryless Shared Last-level Cache. 2012.

[29] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7), 2012.

[30] S. L. Min and J.-L. Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *IEEE Trans. Parallel Distrib. Syst.*, 3(1), 1992.

[31] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*, 2008.

[32] S. K. Nandy and R. Narayan. An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems. 1994.

[33] R. H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Workshop on Parallel and Distributed Debugging*, 1993.

[34] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. SWEL: hardware cache coherence protocols to map shared data onto shared caches. In *PACT*, 2010.

[35] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *PACT*, 2012.

[36] D. Sanchez and C. Kozyrakis. SCD: A scalable coherence directory with flexible sharer set encoding. In *HPCA*, 2012.

[37] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *ISCA*, 1987.

[38] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7), 2010.

[39] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache coherence for GPU architectures. In *HPCA*, 2013.

[40] K. Skadron and D. Clark. Design issues and tradeoffs for write buffers. 1997.

[41] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[42] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: efficient hardware support for disciplined non-determinism. In *ASPLOS*, 2013.

[43] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*, 2008.

[44] D. A. Wallach. *PHD: A Hierarchical Cache Coherent Protocol*. PhD thesis, 1992.

[45] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[46] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad Hoc Synchronization Considered Harmful. In *OSDI*, 2010.

[47] X. Yuan, R. G. Melhem, and R. Gupta. A Timestamp-based Selective Invalidation Scheme for Multiprocessor Cache Coherence. In *ICPP, Vol. 3*, 1996.

[48] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: sharing pattern-based directory coherence for multicore scalability. In *PACT*, 2010.