TC-Release++: An Efficient Timestamp-Based Coherence Protocol for Many-Core Architectures

Yuan Yao, Wenzhi Chen, Tulika Mitra and Yang Xiang

Abstract—
As we enter the era of many-core, providing the shared memory abstraction through cache coherence has become progressively difficult. The standard directory-based coherence does not scale well with increasing core count. Timestamp-based hardware coherence protocols introduced recently offer an attractive alternative solution.
This paper proposes a timestamp-based coherence protocol, called *TC-Release++*, that efficiently supports cache coherence in large-scale systems. Our approach is inspired by *TC-Weak*, a recently proposed timestamp-based coherence protocol targeting GPU architectures. We first design *TC-Release* in an attempt to straightforwardly port TC-Weak to general-purpose many-cores. But re-purposing TC-Weak for general-purpose many-core architectures is challenging due to significant differences both in architecture and the programming model. Indeed the performance of TC-Release turns out to be worse than conventional directory protocols. We overcome the limitations and overheads of TC-Release by exploiting simple hardware support to eliminate frequent memory stalls, and an optimized lifetime prediction mechanism to improve cache performance. The resulting optimized coherence protocol TC-Release++ is highly scalable (storage scales logarithmically with core count) and shows better performance (3.0%) and comparable network traffic (within 1.3%) relative to the baseline MESI directory protocol. We use Murphi to formally verify that TC-Release++ is error-free and imposes small verification cost.

Index Terms—Cache coherence, Many-core architecture, Timestamp-based system, Memory consistency model.

1 Introduction

A considerable consensus has been reached that cache coherence will continue to be employed in future large-scale systems [1] [2]. With the rapid increase in the number of cores on chip, the scalability of a coherence protocol is highly challenging — maintaining coherence across hundreds or thousands of cores will be unprecedentedly difficult. Although directory coherence protocols are currently the de-facto standard, there is growing concern that simply applying the directory coherence to many-core architectures will face serious power and area issues.

An alternative approach to directory coherence are the recently proposed timestamp-based coherence protocols [3] [4] [5] that remove the scalability burden associated with directory coherence. The primary insight behind timestamp coherence is to eliminate the directory for tracking the sharers and instead rely on timestamps to achieve the same effect as invalidations. Timestamp coherence simply assigns a predicted lifetime to each private cache line as it is allocated. A cache line self-invalidates once its lifetime expires. On a write to a cache line, timestamp coherence does not attempt to invalidate the sharers immediately; instead, the write becomes visible when all the private cache copies in the sharer cores have been self-invalidated due to expired lifetime. This scheme eliminates the invalidation traffic and potentially improves performance. Furthermore, the $O(N)$ sharer tracking information (for N cores) in the directory is not required in timestamp coherence, making it much more scalable in terms of area cost, which also translates to energy efficiency.

The principal drawback of timestamp coherence is the overhead due to write stalls. For example, Library Cache Coherence (LCC) [3] — a timestamp coherence protocol — maintains coherence by stalling a write at the L2 cache controller until all the L1 cache copies have expired their timestamps and thus self-invalidated. This write stall is necessary for Sequential Consistent (SC) memory models because all the memory orderings have to be maintained; a write is required to become globally visible before any of the following reads/writes. But relaxed memory consistency models relax some of the ordering requirements. For example, Release Consistency (RC) model [6] relaxes all the memory orderings expect for synchronizations: an acquire guarantees that all the subsequent reads/writes are executed after it and a release guarantees that all the previous reads/writes have completed before it. In other words, RC only requires writes to be visible before a release, and only with respect to the corresponding core that acquires the data protected by synchronization. Thus RC alleviates the need to enforce coherence at every write as long as writes are made globally visible at release points.

TC-Weak [4] leverages this idea to mitigate the write-stalling cost in LCC in the context of GPU coherence where the GPU adopts RC memory model. It achieves this by only stalling on memory fences, ensuring all previously written addresses have been self-invalidated in remote private caches. Inspired by TC-Weak, we implement a similar timestamp-based coherence protocol called TC-Release

- *Y. Yao and W. Chen are with the School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, P.R. China. E-mail: yuanyao, chenwz@zju.edu.cn*
- *T. Mitra is with the School of Computing, National University of Singapore, Singapore. E-mail: tulika@comp.nus.edu.sg*
- *Y. Xiang is with the School of Information Technology, Deakin University, Australia. E-mail: yang@deakin.edu.au*

(Time Coherence at Release) for general-purpose many-core architectures. However, due to significant distinctions between CPU and GPU architectures and the programming models, we find that TC-Release shows subpar performance than a conventional directory coherence protocol. To overcome the disadvantages of TC-Release, we propose TC-Release++ which exploits simple hardware support to eliminate the significant memory stall involved in TC-Release, and an optimized lifetime prediction mechanism to improve cache performance. The resulting coherence protocol's storage per cache line scales logarithmically with core count, and shows better execution time (by 3.0%) and comparable network traffic (within 1.3%) relative to a conventional MESI directory protocol. Additionally, we also use Murphi to formally verify that TC-Release++ is error-free and impose small verification cost.

2 TC-Release

We first present our timestamp-based coherence protocol, called TC-Release, designed for general-purpose many-core architectures. TC-Release is inspired by TC-Weak [4] coherence protocol for GPU architectures. However, we will observe that straightforward re-purposing of TC-Weak for many-core architectures, as we do with TC-Release, incurs significant performance overhead. In the next section, we will propose a number of modifications and optimizations to make TC-Release suitable for many-core architectures.

TC-Weak is a recently proposed timestamp-based coherence protocol for GPU architectures. As mentioned earlier, timestamp coherence assigns a predicted lifetime to each private cache line as it is allocated. A cache line is self-invalidated once its lifetime expires. On a write to a cache line, timestamp coherence does not attempt to invalidate the sharers immediately (in fact the sharer information is not maintained at all unlike directory coherence); instead, the write becomes visible when all the private cache copies in the sharer cores have been self-invalidated due to expired lifetime. To support strict memory consistency model, such as Sequential Consistency, coherence has to be maintained at each write. Thus timestamp-based coherence protocols such as Library Cache Coherence (LCC) [3] stalls every write at the L2 cache controller until all the remote copies have been self-invalidated making the write visible. These write stalls lead to serious performance loss for the protocol.

TC-Weak [4] is based on the principals of Eager Release Consistency implementation, which is previously proposed to reduce communication in distributed shared memory (DSM) systems [7]. The insight is that for relaxed memory models, in particular, Release Consistency (RC) memory model, coherence need not be strictly enforced at every write; making the writes coherent only at release points is sufficient. TC-Weak accomplishes coherence at release point in a core by tracking the largest global timestamp returned by all the writes in the core so far. When a memory fence is encountered (which is indicative of a release point), the protocol requires the memory fence to wait till the largest global timestamp has expired (all remote stale copies have been self-invalidated) ensuring that all the previous writes made by the core have now become globally visible. TC-Weak promises better performance and reduced network traffic than conventional directory protocol for GPU architecture.

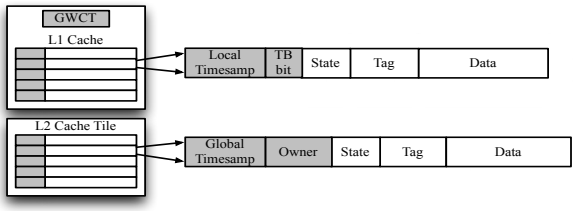


Fig. 1: Hardware extensions for TC-Release.

Our TC-Release (Time Coherence at Release) coherence protocol brings this idea of making writes visible only at release points to general-purpose many-core architectures. However, the difference in architecture and programming model between GPU and general-purpose many-core introduces a number of challenges. TC-Weak uses write-through L1 cache because it performs well for GPU workloads eliminating unnecessary L1 refills of write-once data [4], which is quite common. However, general-purpose CPU workloads show much higher re-use of the dirty lines, rendering a write-back policy more suitable for TC-Release. We observe that L1 re-use of modified data comprises a significant fraction (44.8% on an average) of all L1 read accesses, which is orders of magnitude higher compared to GPU workloads [8]. We also take advantage of the distinction between private and shared data in write-back caches such that private lines do not need to maintain timestamps and self-invalidate upon expiration, leading to higher L1 cache hit rate.

TC-Release assumes private L1 caches and a shared L2 cache, and the L2 cache is physically partitioned into tiles and distributed on chip. Figure 1 shows the hardware extensions for TC-Release. Like LCC and TC-Weak, every L1 and L2 line in TC-Release is augmented with a timestamp. The timestamp in an L1 line (local timestamp) indicates the expiration time of the line, while an L2 line stores the maximum timestamp (global timestamp) of all L1 copies. Similar to TC-Weak, for each L1 cache (i.e., for each core), TC-Release tracks the largest global timestamp returned by the writes to that cache in the Global Write Completion Time (GWCT). TC-Weak maintains one GWCT for each warp in a GPU core. But in TC-Release, we consider simple single-threaded CPUs and only one GWCT is maintained per L1 cache.

Figure 2 shows a simplified example of TC-Release with the execution of the code segment shown at the top of the figure. In the given example, two cores communicate by propagating values of A and B. Initially A and B are both cached in the L1 cache of Core 1 (Line A and B) and have timestamps of 60 and 80, respectively, while the L1 cache of Core 0 does not contain these addresses. Thus the L2 cache lines for A and B also contain timestamps 60 and 80, respectively. At Cycle 20, Core 0 has a write miss at address A and sends a write request to the L2 (❶). Upon receiving the request, the L2 responds with data and a timestamp of 60, corresponding to the expiration time of the copy of Line A's copy cached by Core 1. The L1 cache of core 0 updates its GWCT to 60 upon receiving the response (❷). Similarly, Core 0 performs another write to address B (❸)

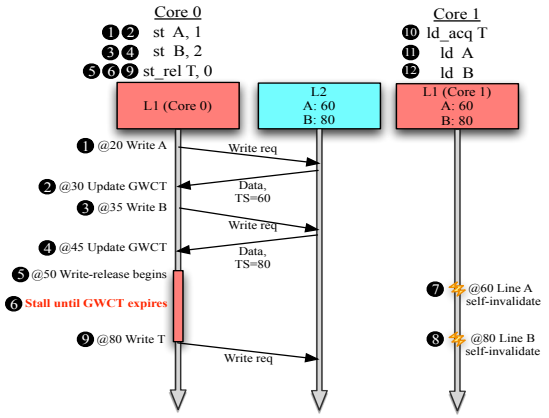


Fig. 2: A simplified example of TC-Release with the execution of the code segment shown at the top.

and subsequently updates the GWCT to 80 (❹), which is the global timestamp of Line B. At Cycle 50, Core 0 executes the store-release instruction to release the synchronization variable T (❺). But as the GWCT at the L1 cache of Core 0 has not expired yet, the cache controller stalls the request until the GWCT expires (❻). At Cycle 60 and 80, Line A and B are self-invalidated in the L1 cache of Core 1 (❼❽). At Cycle 80, Core 0 finally resumes from stalling the write-release and performs the write part of the request, as all previous writes have become globally visible (❾). Finally, Core 1 performs a load-acquire of T (❿) and the following reads to Line A and B (⓫⓬) will get the correct values since their stale copies have been self-invalidated by now, and will obtain values from Core 0.

We now present the detailed protocol design of TC-Release for write-back caches. We distinguish write-release and read-acquire operations from normal writes and reads, as required by the protocol.

2.1 Protocol design

The stable states of TC-Release are similar to a conventional MESI directory protocol as we use write-back policy. The L1 controller in TC-Release has four stable states: Invalid, Shared, Exclusive and Modified, while the L2 controller has Invalid, Shared and Exclusive states. The Exclusive state in the L2 corresponds to both Exclusive and Modified state in the L1. For L1 Exclusive/Modified lines, a pointer is maintained in the L2 line to keep track of the exclusive ownership (as shown in Figure 1). However, the sharing vector for L1 Shared lines are not stored in the L2. As the exclusive ownership is tracked in the L2, L1 Exclusive/Modified lines do not need to maintain timestamps. L2 Exclusive lines may or may not have a timestamp depending on whether there are still unexpired shared copies in the L1 caches.

Write-Release: On a write-release, the L1 controller waits (stalls the write-release request) till its GWCT expires. The stalling guarantees that all the writes before the release have become globally visible. After the GWCT expires, the write part of the write-release is performed as a normal write detailed below.

Normal writes: A normal write hits on L1 Exclusive/Modified lines (Exclusive lines silently transition to Modified). A write misses in the L1 cache for other states and an exclusive request (GetX) is sent to L2. For a write miss, along with data, a timestamp may be returned from the L2 that captures the time when the write will become globally visible.

If the L2 line receiving the GetX request is in Shared state, it immediately responds with data and the global timestamp stored in the line (unlike directory protocol where the other L1 copies have to be invalidated immediately). If the L2 line is in Exclusive state, the request is forwarded to the tracked owner who invalidates its line and sends the data to the requester. Note that it is possible for an L2 Exclusive line to have an unexpired global timestamp as there can still be stale copies lingering around in L1 caches other than the owner. In that case, the timestamp in the L2 line is also transferred in the forwarded request, which is re-forwarded to the original requester by the owner. For an access to an L2 Invalid line, data is loaded from main memory and sent to L1.

Upon receiving the response from the L2, the L1 cache writes the data to its line and transitions to Modified state. To track the global timestamps returned by writes, the GWCT needs to be updated if the response contains a larger timestamp. The L1 completes the transaction by sending an acknowledgment to the L2, which transitions the line to Exclusive state and changes the ownership of the line to the requester.

Normal reads: A normal read hits on L1 lines in Exclusive/Modified state. A read to L1 Shared lines need to check the stored local timestamp: a tag match with an expired timestamp is treated as a read miss, the line is self-invalidated and a read request is sent to the L2. Note that self-invalidating an L1 line due to timestamp expiration does not require explicit events; instead the read to that line is simply treated as a miss after the timestamp expires. A read also misses on L1 Invalid lines and the L2 has to be accessed.

Upon receiving a read request, the L2 will predict a lifetime (i.e., a fixed lifetime value) for the requester if it gets a shared copy of the line. The choice of lifetime value is important as too short predicted lifetime will result in premature expirations and repeated L2 accesses. On the other hand, too long predicted lifetime will require long wait at release points. After every lifetime prediction, the L2 updates the global timestamp of the line to maintain the maximum timestamp among the copies. For an L2 read on Shared lines, the L2 directly responds with the data and a predicted timestamp to the requester. In the case of an L2 read on Exclusive lines, the request with the predicted timestamp is forwarded to the owner, who downgrades its exclusive copy to Shared and changes the local timestamp in the line with the predicted timestamp. The owner then sends the data with the new predicted timestamp to the original requester who updates its data and local timestamp, with a transition to Shared state. A read on L2 Invalid lines gives the requester exclusive ownership, resulting both the L1 and L2 line in Exclusive state.

Read-Acquire: A read-acquire tests if the synchronization variable has been released; otherwise it makes the core to spin-wait until it observes a release performed by another core. A read-acquire in TC-Release can be implemented similar to a normal write (though a read-acquire does not modify data), which gains the L1 line with exclusive owner-

ship. If the acquired synchronization variable has not been released, the core will spin locally in L1 (reading the L1 line again and again) just like a directory protocol. The spin-waiting stops once another core performs a write-release. This is because the core performing the release sends write request to L2 cache, which is forwarded to the core that is spin-waiting because it is the exclusive owner. The spin-waiting core invalidates the line and hence it receives the new value of the synchronization variable on the next read in its spin-wait. This guarantees forward progress in the presence of synchronization.

Evictions: Evictions of L1 Shared lines are silent. An L1 eviction of Exclusive/Modified line needs to inform the L2, which changes the state to Shared (as there can be other stale Shared copies in L1 caches with unexpired timestamp). For L2 evictions, only lines with expired global timestamps can be evicted to maintain inclusion property. Unexpired timestamps are stored in L2 Miss Status Holding Register (MSHR) entries to eliminate stalling on evictions. Note that an eviction of L2 Exclusive line needs to invalidate the owner in L1.

2.2 RC Optimization

In TC-Release, if a release has been observed by the corresponding acquire, the writes before the release are made visible to the acquire core because the acquire core will self-invalidate the stale lines with expired timestamps. However, self-invalidating the lines again before the core performs another acquire is not required. We illustrate this with an example shown in Figure 3, in which two different cores communicate the value of A. In initial state, address A is located in a Shared line in L1 cache of Core 1. As mentioned earlier, the self-invalidation does not explicitly invalidate the copy; instead any line with expired timestamp is considered an invalid line. After Core 1 successfully acquires the synchronization variable T, the first read to A (R1) finds the expired timestamp and self-invalidates the line. R1 then gets the up-to-date value with a predicted timestamp from the L2. Without performing another acquire, Core 1 subsequently encounters another read to A (R2) and finds the newly obtained timestamp expired; but self-invalidating the line again is not necessary because Core 1 has already obtained the up-to-date value from Core 0 (via L2) on the first read of A.

Core 0	Core 1
st A, 1	ld_acq T
st_rel T, 0	**R1**: ld A
	...
	R2: ld A

Fig. 3: Code segment for communication between two cores. Assume there is no acquire between the two loads of A in Core 1.

In order to reduce redundant self-invalidations due to timestamp expirations, we add a Timestamp Bypass (TB) bit per L1 line, as shown in Figure 1. The TB bit of an L1 line is set after its self-invalidation. For a read on L1 Shared lines, the TB bit is examined first before the timestamp check: a read is allowed to hit in L1 when the TB bit is set, bypassing the timestamp check even if it has expired. We call this RC-optimization as it leverages the RC semantics. To ensure the read-acquire → reads/writes ordering, all the TB bits are reset after a read-acquire operation.

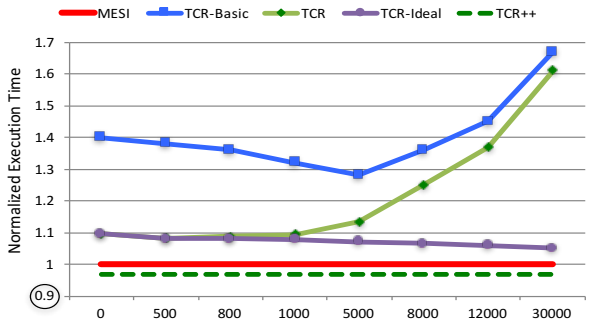


Fig. 4: Normalized execution time of TCR, TCR-Basic, TCR-Ideal with various fixed lifetimes, with respect to baseline MESI directory protocol and TCR++.

2.3 Bottleneck and Trade-offs of TC-Release

To identify the bottleneck of TC-Release, we present a performance characterization of TC-Release with various lifetime values.

Figure 4 shows the normalized execution time of TC-Release with and without the RC-optimization (TCR and TCR-Basic respectively) for increasing values of fixed lifetimes, with respect to baseline MESI directory protocol (red line in the figure). Note that MESI directory protocol does not require timestamp and hence has the same performance throughout. The results are the average of all workloads. As shown in Figure 4, the performance improvement by RC-optimization is remarkable, as it saves a lot of L1 misses. The performance impact of RC-optimization is more significant for small lifetimes, because TCR-Basic suffers from unnecessary L1 misses due to quick timestamp expirations while RC-Optimization protects TC-Release from excessive self-invalidations.

Nonetheless, we can see that TC-Release invariably performs worse than the baseline MESI directory protocol regardless of the different lifetimes used. There are two primary reasons that cause the performance gap between TC-Release and a directory protocol. First, compared to GPU workloads, general-purpose CPU workloads show significantly higher data re-use rate, which requires much larger lifetimes for the L1 lines, making the penalty for memory stall on releases non-trivial. Second, synchronizations in CPU workloads are more fine grained and thus more common, which leads to frequent release-stalling that further exacerbates the performance overhead.

To quantify the performance loss due to stalling on releases, we implement an idealized TC-Release protocol called TCR-Ideal that makes the stalls costless. TCR-Ideal instantaneously invalidates all unexpired L1 lines modified by writes at releases without accounting for timing or traffic, incurring no penalty for release-stalling. In Figure 4, we add the execution time of TC-Ideal with different lifetimes, normalized to MESI. We can see that, with larger lifetimes, the performance difference between TC-Ideal and TC-Release enlarges, as the former is approaching the performance of MESI while stalling on releases deteriorates TC-Release performance.

Interestingly, the performance difference between TC-Release and TCR-Ideal reveals the trade-off between cache

performance and the price paid for release-stalling. On one hand, the high temporal locality of general-purpose CPU workloads requires larger lifetimes. As shown in Figure 4, the performance of TCR-Ideal continuously improves with increasing lifetimes. The performance improvement comes from increased L1 hit rate as larger lifetime reduces misses caused by timestamp expirations. On the other hand, in TC-Release, larger lifetimes can potentially be harmful to the performance as the stalling on releases becomes the bottleneck. In Figure 4, after increasing lifetime from 1000 to 5000 cycles, larger lifetimes in TC-Release begin to show a dramatic downgrading of performance. This is because the substantial performance loss due to release-stalling cannot be offset by the performance gain from the increased cache hit rate.

To make TC-Release adoptable for many-core architectures, its performance gap with directory protocol must be bridged. In the following section, with respect to the trade-off discussed above, we propose TC-Release++, which shows better performance (by 3.0%) than the baseline MESI directory protocol (plotted in dashed green line in Figure 4). TC-Release++ improves TC-Release by mitigating its overheads and provides excellent trade-off in performance, energy and scalability. Note that the performance of TC-Release++ does not change with lifetime values because it does not use a fixed lifetime and instead dynamically predicts the lifetime.

3 TC-RELEASE++

In this section, we present the design of TC-Release++. We first extend TC-Release to save the performance loss due to release-stalling. Then we introduce an optimized lifetime prediction mechanism to meet distinct lifetime values required by different workloads and thereby improve cache performance.

3.1 Eliminating Release-Stalling with Bloom filters

As described earlier, TC-Release lazily makes writes visible to remote cores. However, the propagation of dirty data is still triggered at the writing core (when a release is performed). TC-Release++ takes laziness a step further by delaying the propagation of written data until a remote acquire succeeds. In this respect, TC-Release++ more closely resembles Lazy Release Consistency, which has been proposed for DSM systems to further reduce wasteful communication associated with Eager Release Consistency [7].

Specifically, in TC-Release writes are strictly obliged to be globally visible at a release through the expiration of the GWCT. We relax the write visibility constraint from the time of the release to when another core actually acquires the synchronization variable (that has been released). The idea is to maintain the addresses of the writes that have happened so far; but these writes are not forced to be coherent at a release. Instead, when other cores try to communicate with the release core, they need to check if the address they are trying to read belongs to the set of write addresses (of the release core) and in that case self-invalidate their stale copies.

At release points, we use a Bloom filter to generate a signature at releases that tracks the local writes with unexpired global timestamps. Bloom filter is a space-efficient

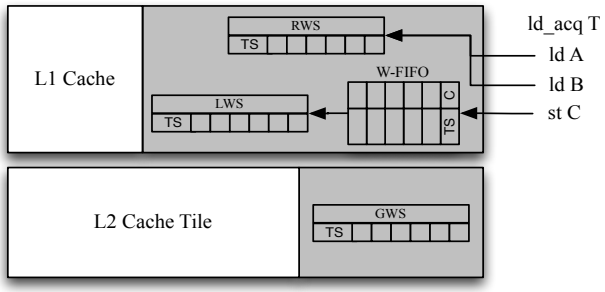


Fig. 5: Hardware extensions for the signature design.

structure to test if a member is in a set, where false positives are possible but false negatives are not permitted. On an acquire, the L1 cache obtains the corresponding signature and for subsequent reads in Shared state, the requested line is self-invalidated if the address hits in the signature even if the timestamp of the line has not expired. By keeping track of uncompleted writes before release and selectively self-invalidating stale lines, the heavy burden of release-stalling is effectively removed.

In our timestamp-based coherence protocol TC-Release++, using Bloom filter for write-tracking has a big advantage: the signature naturally inherits a timestamp from the coherence protocol, indicating the global completion time of the tracked writes. When the timestamp of the signature expires, the filter field (a bit-vector) can be cleared because all the writes tracked in the signature have become globally visible. We call this operation signature clear. All signatures in our proposal have the same structure: the filter field and a timestamp that indicates the signature's expiration time. We also provide a detailed example of how the signatures are cleared in the supplemental material.

3.1.1 Hardware extensions and protocol design

Figure 5 shows the hardware extensions for the signature design. Conceptually, in every L1, the Local Write Set (LWS) signature tracks the locally completed yet not globally visible writes, and the Remote Write Set (RWS) signature contains the write-set created by other (remote) cores. The Global Write Set (GWS) is maintained per L2 tile, and behaves as the intermediary for signature communication. We now explain the hardware structures with detailed operations. Detailed state transition tables of TC-Release++ can be found in the supplemental material.

Normal writes: Identical with TC-Release, normal writes hit on Exclusive/Modified lines in the L1. For an L1 write miss that returns a timestamp, an entry is enqueued at the tail of a write FIFO (W-FIFO), as shown in Figure 5. The entry is constructed by combining the write address with the returned timestamp. In the example code segment shown in the right side of Figure 5, the write to address C returns a global timestamp from the L2 and therefore enqueues a new entry to the W-FIFO. If the entry reaches the head of the W-FIFO, it will replace the old entry at the head. If the replaced entry has an unexpired timestamp, the address is inserted into the LWS. The LWS will also update its timestamp if the replaced entry has a larger timestamp. For an insertion to the LWS, the signature is cleared first if its timestamp has expired. With the help of the W-FIFO, the size of write-set tracked in the LWS is reduced.

Write-Release: On a write-release, the L1 controller triggers a W-FIFO flush signal that dequeues the W-FIFO until it reaches the head. Every evicted entry with unexpired timestamp inserts its address into the LWS and updates the timestamp of the signature. After the W-FIFO flush completes, the L1 will send a release request (REL) containing the LWS to the appropriate L2 tile, according to the address of the released synchronization variable. Note that if the RWS in the L1 has not expired, the protocol will first perform an union of the RWS with the signature in the REL. This guarantees the transitivity property some programs may rely on [2]. The timestamp of the signature will be the maximum of the LWS and RWS timestamps, which also applies to other signature unions discussed later.

The L2 tile, upon receiving the REL, unions the received signature with the Global Write Set (GWS) signature. Note that a signature clear is performed in the GWS first if it has an expired timestamp. The L2 then sends an acknowledgment to the requester, signaling the L1 to proceed to the write part of the release operation, which is treated as a normal write.

Read-Acquire: For a read-acquire, in order to make all writes preceding the corresponding release visible to the acquire core, it needs to obtain the relevant signature in the L2. As mentioned earlier, a read-acquire may spin locally from L1 if the synchronization data is still held by another core, which may result in repeated L2 accesses for obtaining the signature. To address this issue, we introduce two new stable states Exclusive_A and Modified_A in the L1 controller, distinguishing normal private lines from those involved in spin-waiting. A read-acquire on L1 lines in these two states is not required to obtain the signature. A normal read or write will hit on L1 lines in Exclusive_A/Modified_A, with normal writes transitioning the line to Modified. The added two states also help to reduce the Timestamp Bypass (TB) bits resets in the RC-optimization (discussed in Section 2.2), as a read-acquire involved in spin-waiting does not need to reset the TB bits. Detailed operations are discussed below.

A read-acquire misses on L1 Invalid or Shared lines, and an acquire request (ACQ) is sent to the L2 tile based on the address of the acquired synchronization variable. An ACQ is similar to a GetX, with the difference that the L2 also needs to transfer the GWS in the exclusive data response. After the L1 receives the response, the L1 line transitions to Exclusive_A state.

A read-acquire can hit on L1 Exclusive/Modified lines, but an ACQ must be sent to the L2 first, as the synchronization data may have been released but subsequently fetched to the L1 by normal reads/writes. Since the L1 is the current owner of the line, in this case the L2 only needs to respond with the GWS (and no data is transferred). After receiving response from the L2, the read hits in the L1 and transitions the line from Exclusive or Modified to Exclusive_A or Modified_A, respectively.

L1 lines in Exclusive_A or Modified_A allow a read-acquire to hit locally without sending an ACQ to the L2, as the core is probably spin-waiting. The L1 line will be eventually invalidated by a release, hence a legitimate ACQ will be sent for the following read (within read-acquire spinning) to the Invalid line.

When the L1 receives the response for an ACQ, The L1 unions the obtained signature to the RWS, which will be checked for subsequent normal reads on Shared lines.

Normal reads: In TC-Release, a normal read hits on L1 Shared lines with an unexpired timestamp. In contrast, TC-Release++ also needs to check the RWS signature to determine if the data has been modified by a remote core. As shown in the example in Figure 5, the reads to A and B are required to consult the RWS. If the address hits in the signature, the line is self-invalidated and a read request will be sent to L2. On a check of the signature, a signature clear is performed if possible. Operations for a normal read on other L1 states are the same as TC-Release.

The usage of Timestamp Bypass (TB) bit in TC-Release can be easily extended to TC-Release++. For a read on L1 Shared lines with the TB bit set, the read is considered as a hit and the checks on both the timestamp and the signature are bypassed.

3.2 Lifetime Prediction and Shared Read-Only Optimization

It is important to highlight the trade-offs in lifetime prediction before we describe our prediction mechanism. Basically, the lifetime needs to be long enough to take advantage of the high data re-use in the workloads. However, unnecessarily large lifetime may increase the lifetime of a signature, consequently degrading performance due to increased false-positive matches in the bloom filter. To exploit the observations made in the previous subsection, we take access patterns into account for lifetime prediction. We categorize shared cache lines into four types: Write-frequent lines are vulnerable in the L1 cache, hence short lifetime should be enough to accommodate them. Some Read-frequent lines have moderate re-use rate and are likely to favor medium lifetimes. Read-frequent lines have greater tendency to stay longer in L1 caches for further re-use, requiring long lifetimes. In addition, we introduce another state *SharedRO* for shared lines with read-only behavior to take advantage of the significant percentage of accesses to the shared read-only lines,. The SharedRO lines do not have timestamps that dictates the expiration time for the lines, essentially behaving as lines with infinite lifetime.

Instead of using a single lifetime value as proposed in TC-Weak, we maintain three lifetime values for different access patterns described above (SharedRO lines do not require a lifetime value). To extract the access pattern at runtime for the lifetime predictor, we exploit the owner bits in L2 lines to record the read frequency of the line, as the owner bits are not used for L2 lines in Shared state. A read to an L2 Exclusive line will make it transition to Shared state with the read counter initialized to zero. Every subsequent L2 read to a Shared line due to L1 timestamp expiration will increase the read counter by one. When the read counter exceeds a predefined threshold, the access pattern is deemed changed and the next level lifetime value for higher read frequency will be used for lifetime prediction. When the read counter exceeds the last threshold, the Shared line transitions to SharedRO state. To adjust the lifetime value within one particular access pattern, a read will increase the lifetime value by a fixed amount t_R (if it does not exceed the lifetime value for the next level). Similarly a

write or an eviction of an unexpired lines will decrease the corresponding lifetime value by t_W.

A write request to SharedRO line triggers a broadcast of invalidation requests and subsequent acknowledgments from the L1 caches. Our simulations results show that such ShardRO mis-prediction induced invalidations are extremely rare — only about one in every ten thousand shared writes involves an invalidation broadcast.

In a real system, it is non-trivial to accurately predict the lifetime of a cache line. Here, we build a simple predictor based on reference patterns in the L2 cache. It is possible to do this more accurately with other metrics. For example, the live time metric proposed in Timekeeping [9] can be useful in our lifetime prediction, such explorations are left for future work.

3.3 Timestamp rollover

In TC-Release++, the global time counters monotonically increase and may roll over. Here we propose one possible solution to this issue.

For every lifetime prediction in the L2, we do not predict any timestamp that rolls over. Specifically, if the current time added by the predicted lifetime causes rollover, the maximum timestamp (that does not roll over) is predicted. This guarantees that when the global time counters roll over, every timestamp in the system should be expired. The expired timestamps can be in cache lines, W-FIFOs, signatures and network messages. We will discuss how to deal with these expired timestamps when rollover happens.

First, when the global time counters roll over, all the L1s stop receiving requests from cores, so no more requests can be issued to the cache hierarchy. Conservatively, we wait 10k cycles to make sure there is no outstanding network messages. During this period of time, if the L2 receives any read request, it predicts an expired timestamp (e.g., a timestamp of 0) for the request. After sinking the network messages, we flash-invalidate the L1 Shared lines since they are expired. Simultaneously, all the W-FIFOs and signatures are cleared, because writes tracked in either W-FIFOs or signatures have become globally visible. Finally, the L1s are resumed from blocking requests from cores. Note that 10k cycles of L1 stalling seems expensive but it happens fairly infrequently and therefore is acceptable (e.g., for a 32-bit timestamp, 10k in every 2^{32} cycles is negligible).

3.4 Memory consistency

RC has been adopted at least partially by ARM [10], Alpha [11], and Itanium [12] because it is adequately weak for many hardware designs, but strong enough to reason easily about data races [13]. A processor supporting RC provides SC where the software guarantees data-race-free [14].

The RC model TC-Release models is RCsc, one variant of RC [6]. Like other variants of RC, it defines acquire and release semantics for synchronization. These semantics order reads with respect to an acquire (read-acquire $\rightarrow$ reads) and writes with respect to a release (writes $\rightarrow$ write-release). In addition, it requires synchronization accesses to be sequentially consistent.

Synchronization accesses are coherent and sequentially consistent among themselves in our coherence protocols. Both TC-Release and TC-Release++ track and serialize accesses to each synchronization variable. This is accomplished by obtaining exclusive ownership for every synchronization access. Thus, they are coherent because cores cannot observe different write orders, only the order in which writes register their ownership in the L2. They are SC because, in addition to the above, read-acquires and write-releases are blocking in the core so no later synchronization operations can be initiated until they complete.

4 Verification

We formally verify TC-Release++ described above using the explicit-state model checking tool Murphi [15], a widely-used tool for verifying cache coherence protocols.

4.1 Abstract model

We adopt the standard method for debugging cache coherence protocols: we build the abstract model of the TC-Release++ protocol detailed in Section 3 and perform an exhaustive state enumeration of the model for a small-sized configuration [17].

To limit the state space, we leverage the symmetry reduction in Murphi by modeling processors, addresses and data values as *scalarset* (a data type in Murphi), as the operations involving these structures do not depend on the ordering of the elements. A core is modeled as an array of cache entries consisting of L1 state information along with protocol-specific fields like the local timestamp and the Timestamp Bypass (TB) bit for TC-Release++. Similarly, L2 is also modeled as an array of cache entries, each with L2 state information, an owner pointer and other protocol-specific fields like the global timestamp. The on-chip network is modeled as an unordered buffer. The Bloom filters are modeled as perfect filters which incur no false-positive hits. False-positives are modeled in full-system simulation on gem5 [18], which we will show in Section 6.

The above strategies reduce the state space, nevertheless, the fundamental differences between TC-Release++ and a conventional coherence protocol (i.e., eagerly providing SC, without timestamps, etc.) renders canonical abstract models difficult to be straightforwardly reused (e.g., the DASH protocol model shipped with the Murphi release). The rest of this subsection highlights the challenges of verifying TC-Release++, followed by our solutions.

4.1.1 DRF model and DRF-relaxed model

In contrast to a coherence protocol that strictly models SC like the baseline MESI directory protocol, TC-Release++ models RC, providing SC where the program is free of data races (DRF [14] or properly labeled [6]). On the other hand, though our protocol guarantees correctness for DRF programs, as data races can still occur in software, we also need to verify that TC-Release++ introduces no deadlock or livelock for racy programs. To this end, we conduct two versions of verification using Murphi: one models DRF guarantee from software whereas the other relaxes it.

DRF model: To model the data-race-free guarantee from software for TC-Release++, our DRF-based verification makes the following efforts.

First, cores need to provide ordering guarantee for synchronization accesses, which are sequentially consistent and

also enforce a downward or upward fence. This is achieved by 1) a core cannot issue a synchronization access until it completes all the pending memory requests; 2) a core is stalled from issuing any more memory requests until the current synchronization access is completed. In Murphi, this is implemented simply by checking the L1 request queue of the core.

Second, to correctly model acquire and release, we need to define when an acquire is deemed successful and when a core is allowed to issue a release. In our model, every synchronization variable can be one of three values: the initial *undefined* value, *AcquiredValue* and *ReleasedValue*. Upon receiving the data response for an acquire, if the value of the data is *ReleasedValue* or *undefined*, the acquire is successful. Otherwise, another core has not released the synchronization variable, so our verification will model the spin-waiting behavior of the acquire core. Once the core successfully acquires the synchronization variable, it modifies the data value to *AcquiredValue*. In addition, every core records the synchronization variable it has acquired. A core can perform a release to an acquired synchronization variable, which modifies its data to *ReleasedValue*.

Third, cores can generate normal read/write requests only when they are properly synchronized. Specifically, a core needs to acquire the corresponding synchronization variable before it can access the data propagated by the synchronization. To prevent the core from issuing racy accesses, if a core has not acquired a synchronization variable, it always acquires one before issuing any normal reads/writes. In addition, for every synchronization variable, we implement a set of addresses that it should propagate, which we call the *guard* set. After a core has acquired a synchronization variable, it can 1) issue reads or writes to addresses in the obtained guard set, or 2) an unaccessed address which is not in the guard set of any synchronization variable. In the second case, the address will be added to the guard set of the acquired synchronization variable. Figure 6 illustrates our scheme. As shown in Figure 6, Core 0 acquires synchronization variable A (❶) and attempts to write address x. Since x is unaccessed (not in any guard set), this write is race-free and x is added to the guard set of A (❷). Then Core 0 releases A (❸). Similarly, Core 2 acquires B (❹), writes y and adds it to guard set of B (❺), before it releases B (❻). As A has been released by Core 0, Core 1 successfully acquires A (❼). Core 1 is allowed to access x (❽) and z (❿), which are in A's guard set and unaccessed, respectively. However, access to y (which is in B's guard set) is not guaranteed in the model since it violates the data-race-freedom guarantee (❾). Note that our model does not generate programs using fine-grained locks (e.g., nested locks). Fine-grained locking is error-prone as it can lead to deadlock [19]. Generating error-free fine-grained locks in Murphi is out of the scope of our verification. Also note that before a release, with respect to other cores, our DRF model only orders accesses after the acquire, because it does not allow a core to generate normal accesses until it performs a successful acquire. In this sense, it makes our modeled consistency somehow weaker than RC, which orders all normal accesses before a release ([6], condition 3.1, A and B).

Finally, to satisfy the data correctness invariant (which will be discussed in Section 4.1.3), we also record the last written value for every address (including synchronization addresses). This will be check against the received data for every read request.

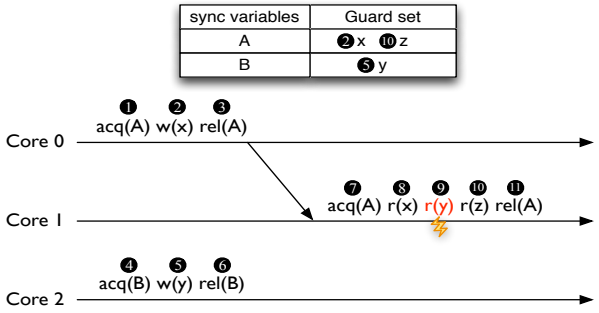


Fig. 6: Data-race-free guarantee.

TABLE 1: Dekker's algorithm (the code for priority control when both cores intend to enter CS is omitted).

Core 1	Core 2

R1: st_rel x, 1
A1: ld_acq r1, y
beqz r1, CS1
// priority control
...
jmp A1
CS1:
// critical section
	R2: st_rel y, 1
	A2: ld_acq r2, x
	beqz r2, CS2
	// priority control
	...
	jmp A2
	CS2:
	// critical section

Initially, x = y = 0.	

Particularly, our verification model also tests the SC behavior among synchronization accesses. For example, our model will emit and test Dekker's algorithm shown in Table 1, which is a legal DRF program and should revert to SC behavior (r1 and r2 cannot both read 0, thus at least one core will enter the pooling loop, guaranteeing mutual exclusion for the critical sections). We first explain that our coherence protocol prohibits the infeasible states (both r1 and r2 are set to 0) after executing release/acquire in Dekker's algorithm on two cores in parallel (in any order), then we show that our verification model allows the program to be emitted.

As aforementioned in Section 3.4, synchronization accesses in TC-Release++ need to obtain exclusive ownership (similar to [20] [21]), thus cores cannot observe different write orders, only the order in which writes register their ownership in the L2. Also, acquires and releases are blocking in the core so no later synchronization accesses can be issued until they complete. With respect to the registration order of acquire/release in Table 1, Figure 7 illustrates three execution sequences of them. The registration order dictates the order cores see the writes. For example, in Figure 7a, the outcome is (r1, r2) = (0, 1) because A1 registers before R2 while A2 registers after R1, so A1 will read the initial value 0 while A2 can see the written value of R1. Similarly, the outcomes of Figure 7b and Figure 7c are (r1, r2) = (1, 0) and (1, 1), respectively. Note that Figure 7c depicts only one of the four possible execution sequences that lead to (r1, r2) = (1, 1); this execution is {R1, R2, A1, A2}, and the others are {R1, R2, A2, A1}, {R2, R1, A1, A2}, and {R2, R1, A2, A1}. Thus, across all possible six execution sequences, our protocol does not produce the non-SC outcome where (r1, r2) = (0, 0).

Our DRF model will emit Dekker's algorithm, because 1) the modeled spin-waiting behavior guarantees normal accesses be emitted after a successful acquire, but it imposes no constrain on emitting synchronization accesses. There-

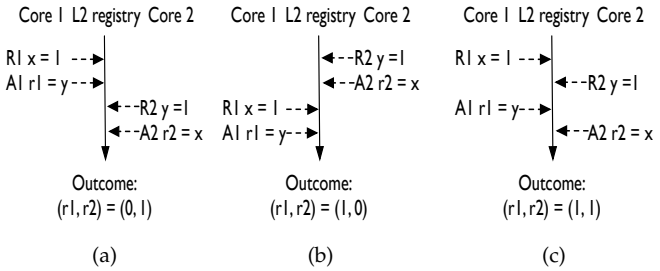


Fig. 7: Execution sequences of Dekker's algorithm

fore, the acquires and releases in Table 1 are allowed to be emitted. 2) Similarly, the introduced guard set prevents normal accesses to racy addresses, which does not affect synchronization accesses. That is, A1 and A2 are considered as race-free and therefore can be emitted in our model. 3) Though we do not handle nested locking, Dekker's algorithm is still tested as it does not contain any nested locks (new acquires A1 and A2 are performed after releases R1 and R2, respectively).

DRF-relaxed model: For the DRF-relaxed verification, we let Murphi generate racy accesses (the third scheme of the DRF model is not used). As correctness is not guaranteed in this case, we do not check the last two invariants in Section 4.1.3 (i.e., Bloom filter correctness and data transfer correctness). In other words, this verification is only used for proving that our protocol imposes no deadlock/livelock under racy programs.

4.1.2 Timestamp abstraction

One major challenge in verifying our coherence protocols is to avoid the state explosion problem [22] caused by timestamp modeling.

If we simply model a timestamp as a counter, in an explicit state model checker like Murphi, any change to the timestamp value will result in a new state. For example, considering the case when all the L1 lines have expired, monotonically incrementing the global counter will consequently lead to an infinite-state system.

However, this is unnecessary, as the behavior of a timestamp is limited to check if it has expired by comparing it to the global clock. Therefore, the operations on the timestamps depend on, rather than their absolute values, the relative order of the timestamps. That is to say, if change to a timestamp does not alter the relative order of them, it has no effect on triggering any timestamp induced events in our protocols, thus no new system state should be generated.

In order to achieve state reduction, we implement a version of timestamp abstraction scheme similar to [23] [24] which exploits the relative order of the timestamps. Let $\{TS_1, TS_2, TS_3, \cdots, TS_n\}$ be the set of all timestamps in the system. Instead of modeling timestamps as counters, we map the timestamps $\{TS_1, TS_2, TS_3, \cdots, TS_n\}$ into consecutive integers $\{1, 2, 3, \cdots, n\}$. For instance, if we have 5 timestamps of {1, 2, 5, 8, 10}, they will be reduced to {1, 2, 3, 4, 5}. Specifically, every rule fired by the model checker that causes change to the timestamps (e.g., lifetime prediction, global clock tick, etc.), a function (implemented in less than 100 lines of Murphi code) will be invoked to adjust the relative representation of the timestamps. The function sorts the timestamps of the system, and then assigns them the new relative values.

For lifetime prediction in the L2, we model a simple static $\Delta = 1$ algorithm which predicts a timestamp as the global clock added by one. Expired timestamps are denoted as *undefined* state in Murphi, so they will be excluded from state enumeration. We also model global clock tick by having a rule that increments it by one.

Note that more sophisticated methods like translating the timestamp-based system into timed automata [25] can be used to address the infinite-state issue, but in this work we use a simple state reduction scheme.

4.1.3 Invariants

As maintained earlier, TC-Release++ lazily makes writes visible to other cores, therefore the conventional invariants (e.g., sharer vector based checks) used for write-invalidating protocols cannot be used. Instead, we develop four invariants to verify our protocol.

Only one exclusive copy exits. As our protocols tracks the exclusive ownership in the L2, there can exist only one L1 copy (the owner) in Exclusive/Modified states. This invariant checks the exclusivity of L1 modified data, any violation to it can obviously lead to data inconsistency.

Timestamp validity. If an L2 line has an expired timestamp, then all its L1 shared copies should have expired. This invariant checks that the global timestamp in the L2 is correct, which always tracks the largest timestamp of the L1 copies.

Bloom filter correctness. This invariant is only used for the DRF model. If the value of a synchronization variable is *ReleasedValue*, for every address in its guard set, if the value of any L1 copy is different from the recorded last written value, then the address must hit in the Bloom filter in the L2. This is because an L1 line in Shared state can potentially be stale, but when the core that modifies the data has performed the release, the Bloom filter stored in the L2 must track the written address. This invariant checks that after a core leaves the critical section and performs a release, the Bloom filter is correctly generated.

Data transfer correctness. This invariant is only used for the DRF model. To check the data correctness of the protocols, upon data receipt of a read request, the received data will be compared to the recorded last written value of the address. This invariant checks if there is any data error in communication among cores, which is of crucial importance to the correctness of the TC-Release++ coherence protocol.

4.2 Verification results

We run Murphi on a Xeon E5620 CPU with 32GB of memory limit, with Murphi's -c compilation option to use hash compaction of the state descriptor. Thanks to our state reduction strategies, with a configuration of 3-core model, the DRF model checking process reaches 22.1 million states in 1200 seconds. For the DRF-relaxed model, it reaches 91.1 million states in 1.4 hours. The DRF-relaxed model generates more states because it allows racy programs to be tested. Verification results reveal that TC-Release++ does not violate any of the conceived invariants, introduce deadlocks or livelocks or have race conditions that violate correctness.

5 Methodology

In this section, we provide the simulation infrastructure and workloads used to carry out our evaluation.

5.1 Simulation Environment

For evaluation of our proposal, we use the gem5 full-system simulator [18] with Ruby memory system enabled. We simulate A 64-tile 2D mesh network-on-chip. Table 2 lists detailed parameters of the simulated system. We do not simulate more than 64 cores because gem5 currently only supports full-system simulation for up to 64 cores. We choose Alpha ISA with minor ISA extension to explicitly provide acquire and release semantics for the hardware (see Section 5.2 for details). We use the H3 Bloom filter implementation, with four hashing functions and a 256-bit filter. We use H3 for its low-complexity and easy implementation. The chosen size of the Bloom filter offers a good compromise between the hardware overhead and the reduction in the number of Bloom filter false-positive hits. Likewise, we determine the W-FIFO size to be 16 entries. The size of the timestamp used in our simulation is 32-bit.

The baseline protocol used in our evaluation is the MESI directory protocol shipped with gem5, where the directory information is embedded in the LLC (last-level cache, L2 in this case) tags. A full-map sharing vector (i.e., 64-bit in our case) is stored in every LLC entry to precisely track the sharers.

TABLE 2: Simulation parameters.

Cores	64 in-order cores at 2 GHz, Alpha ISA, single-thread, IPC-1 except on L1 misses
L1 Cache	Split I & D, 32KB, 4-way, 64B cacheline, LRU, 2-cycle access latency
L2 Cache	Shared, 32MB (64 slices of 512KB each), 16-way, 64B cacheline, LRU, 9-cycle access latency
Network	2D Mesh, 8 rows, 16B-flit, 1/5-flit control/data packets
Memory	2GB, DDR3, 16 channels
Timestamp size	32 bits
Bloom filter	256-bit filter, 4 H_3 hashing function
W-FIFO size	16 entries

TABLE 3: Workloads and input size.

PARSEC	blackscholes	simmedium
	bodytrack	simsmall
	ferret	simsmall
	fluidanimate	simsmall
	swaptions	simsmall
SPALSH-2	barnes	16K particles, ts=0.25
	ocean_cp	514x514 Grid
	radiosity	BF refinement=1.5e-1
	raytrace	Teapot
	water_nsqured	15^3 molecules
	water_spatial	15^3 molecules
	fft	4M points
	lu_cb	512x512 matrix, block=16
	lu_ncb	512x512 matrix, block=16
	radix	16M keys, radix=4K

5.2 Workloads

We use PARSEC [26] and SPLASH-2 [27] workloads to evaluate our proposal. Table 3 shows the 15 workloads and input size used in simulation. For stable and faithful measurements, we run each experiment multiple times and bind each thread to a particular core by invoking the Linux system function *pthread_setaffinity_np* when the threads are spawned. All workloads run correctly to completion, and the statistics are collected from start to the end of the parallel phase. To obtain the acquire and release semantics from the applications as required by our proposal, we extend the Alpha ISA with special read-acquire and write-release instructions and instrument the libraries used as synchronization primitives in the workloads so that they are exposed to the hardware architecture.

TABLE 4: Storage requirements for TCR++ in an N-core system.

TCR	**Per L1/L2 line:** Timestamp, 32-bit Timestamp Bypass bit (L1 line only), 1-bit Owner pointer (L2 line only), $\log_2(N)$-bit **Per L1:** GWCT, 32-bit
Signature design	**Per L1:** RWS/LWS, 256-bit filter + 32-bit timestamp = 288-bit W-FIFO: 16 entries * (32-bit for addr + 32-bit timestamp) = 128B **Per L2 tile:** GWS: 256-bit filer + 32-bit timestamp = 288-bit
Lifetime prediction	**Per L2 tile:** Lifetime values, 3 * 32-bit for each = 96-bit

6 Evaluation

In order to evaluate our proposal, besides the baseline MESI directory protocol, we present detailed results for four configurations. TCR-Basic is similar to TC-Weak but with necessary adaptations for general-purpose many-core architectures as discussed in Section 2. TCR adds the important RC-optimization on top of TCR-Basic. TCR++ improves the basic TCR protocol by applying techniques detailed in Section 3 that reduces the stalls at release points and performs better lifetime prediction. As an ideal reference design, we also implement an infinite size bloom filter with TCR++ and we denote this idealized configuration as TCR++Inf.

As TCR-Basic and TCR use fixed lifetime prediction, we select the value to be 4,500 cycles and 900 cycles, respectively, because these values yield the best performance. Larger lifetime values begin to degrade performance with increasing stalls at release points. We find that static lifetime for TCR-Basic and TCR performs better than dynamic lifetime prediction proposed in TC-Weak [4] because dynamic lifetime prediction attempts to accommodate the high L1 data re-use rate, which results in longer lifetime and suffers more from stalls at release. The initial values for the three lifetimes used in TCR++ are 10K (write-frequent), 85K (moderate read-frequent) and 160K cycles (read-frequent) and the respective thresholds for read-counter to upgrade the access patterns are 16 (upgrade to moderate read-frequent), 32 (upgrade to read-frequent) and 64 (upgrade to shared read-only). We determine the lifetime values as they evenly divide the re-uses of shared read/write lines. The lifetime adjustment values t_R and t_W used within each type of access pattern are 16 and 256 cycles, respectively.

In the following subsections, we first assess the hardware storage required by TCR++ and compare it to conventional directory coherence. Then we validate our proposal by presenting detailed simulation results of execution time, network traffic and cache performance.

6.1 Storage overheads

Table 4 shows the storage requirements for TCR++. The per line storage requirement for maintaining the timestamp has the most significant impact on hardware cost. The additional storage overheads for implementing the proposed signature design and lifetime prediction is modest as it does not require any per line cost, adding up to less than 1% of storage for the per line timestamp.

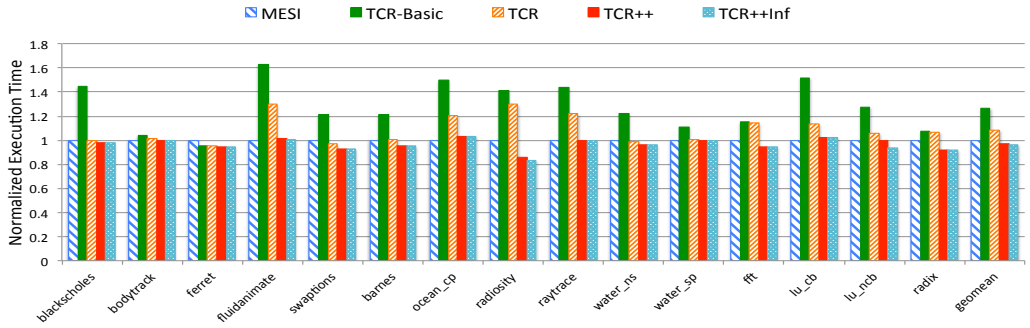


Fig. 8: Execution time of all configurations, normalized to MESI.

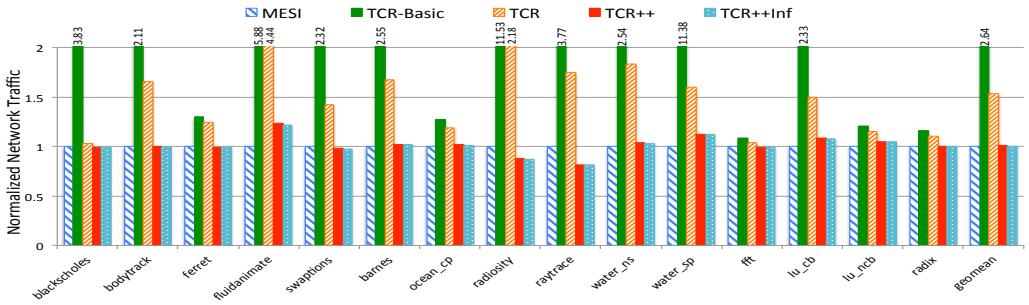


Fig. 9: Network traffic of all configurations, normalized to MESI.

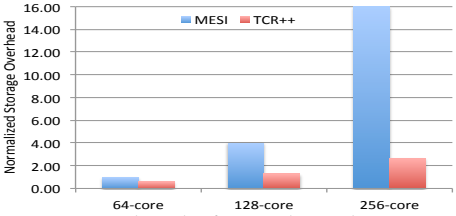


Fig. 10: Storage overheads for cache coherence in TCR++ and MESI, with up to 256 cores.

As aforementioned, a plethora of works have investigated minimizing the directory footprints [28] [29], here we choose a standard full-map directory as our baseline. Compared to the baseline directory protocol, TCR++ only requires $O(\log N)$ storage per line for an N-core system rather than $O(N)$ directory information. Figure 10 shows the coherence storage overheads of TCR++ and MESI for up to 256 cores. We can see TCR++ is significantly more scalable, reducing as much as 83% of the coherence storage overhead compared to MESI at 256 cores. Compared to the chip caches, the storage overheads of TC-Release++ are less than 7.2% of the storage of the private cache plus the shared L2 at 256 cores.

We do not provide a detailed study of area benefits from the $O(\log N)$ coherence storage of TCR++ as it has been well reasoned in [1]. When the on-chip core count grows radically, say to 256 cores, the storage of a full directory will require 256-bit sharer vector per LLC cache line, which equals to 50% of the whole LLC storage for 64B cache line. Moreover, the LLC occupies a considerable portion of the chip area (as much as 50% in modern chips [30]). As illustrated in Figure 10, TCR++ reduces the directory storage overhead by 83% compared to MESI at 256-core, which can directly translate to significant savings in chip area.

6.2 Performance results

Figure 8 and Figure 9 show the execution time and network traffic for all the workloads for the five configurations, normalized to the baseline MESI with directory. To further evaluate the impact of our proposal on cache behavior, we plot the normalized L1 miss rate (w.r.t. MESI with directory) and the breakdown of L1 hits for all evaluated configurations in Figure 11 and Figure 12, respectively.

TCR-Basic and TCR: On an average, TCR-Basic shows 26.6% slowdown compared to the baseline MESI. The best case, `ferret`, performs 4.7% faster than the baseline, while the worst case has a slowdown of 63.2% for `fluidanimate`. Benefiting from the RC-optimization, TCR is able to speed up TCR-Basic by 14.2%. Three workloads (`ferret`, `swaptions` and `water_nsqured`) show slightly better performance compared to MESI, while the worst case performance (`fluidanimate`) is still 30.0% slower than the baseline MESI. The speedup of TCR over TCR-Basic primarily results from the significant reduction in L1 misses due to the RC-optimization (see Figure 11; on an average, TCR has 50.1% decrease in L1 miss rate over TCR-Basic).

Nonetheless, on an average, TCR still performs 8.6% worse than MESI. The main reason behind the subpar performance of TCR is the performance penalty for stalling on releases, and the performance loss gets exacerbated in case of frequent synchronizations (e.g., `fluidanimate` with the worst case performance). Moreover, substantial memory stalls on releases prohibits larger lifetime values, which in turn hampers the L1 cache performance. Consequently, TCR shows an average increase of 35.7% L1 miss rate over MESI. The high percentage of shared reads in `radiosity` suffers from timestamp expirations, causing 203% more L1 misses than the baseline, as shown in Figure 11. The significant increase in L1 miss rate also affects the generated network traffic. As we can see in Figure 9, TCR has an average increase of 53.2% in network traffic over MESI.

TCR shows worse performance for workloads with lots of shared data accesses and frequent synchronizations. It reveals mediocre performance for workloads with small shared data working set and predominant accesses to pri-

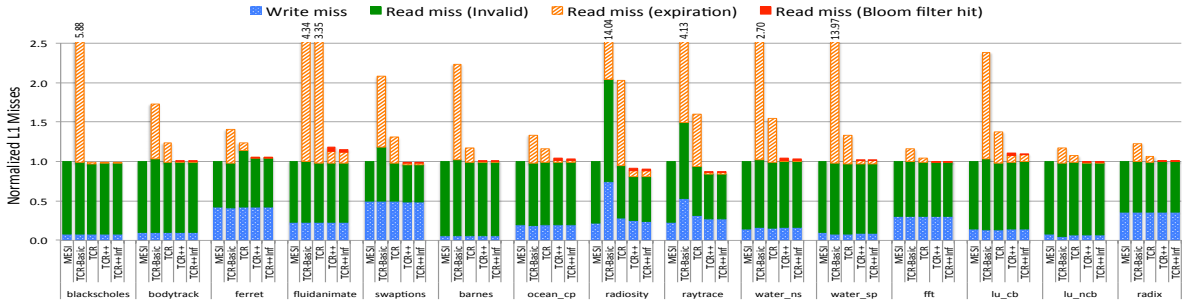


Fig. 11: L1 miss rate of evaluated configurations, normalized to the baseline MESI. Misses are broken down by writes and reads, with the latter split up by three causes: Invalid state, lifetime expiration, and Bloom filter hit.

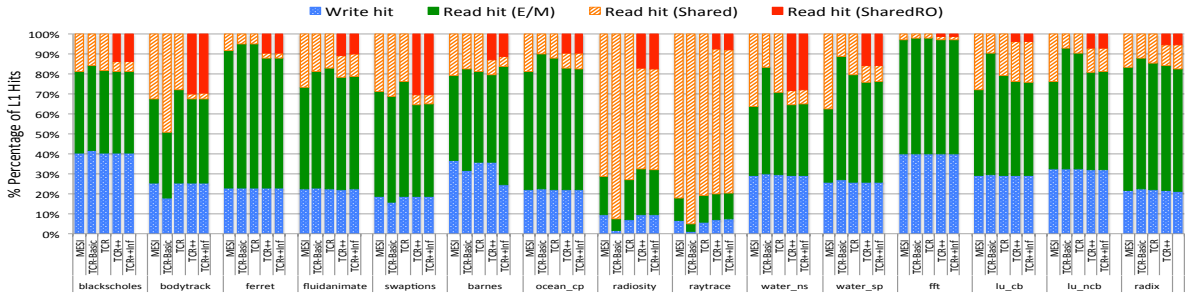


Fig. 12: L1 hits breakdown by writes and reads, with the latter split up by cache states: Exclusive/Modified, Shared and SharedRO.

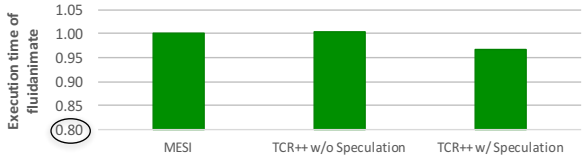


Fig. 13: Performance improvement by speculation.

vate data. For example, `fft`, `radix` and `ferret` are less sensitive to release-stalling because more than 80% of L1 hits are to temporarily private states (Exclusive/Modified), referring to Figure 12.

TCR++: By relaxing the write visiblity time from a release to the corresponding acquire, in tandem with the optimized lifetime prediction, TCR++ is rewarded with an average of 10.7% speedup over TCR. Compared to the baseline MESI, TCR++ is on an average 3.0% faster. The best cases, `radiosity` and `radix`, perform 14.0% and 8.3% better than the baseline, respectively. The worst case is `ocean_cp` with 3.3% slowdown. TCR++ shows comparable or better performance than MESI because of its faster writes as shared lines are not explicitly invalidated and acknowledged as in directory coherence protocols. As the writes can complete faster, the cache line stays in the blocking state for shorter duration, making the subsequent reads to the line faster.

In contrast to TCR with fixed lifetimes, TCR++ is able to fully utilize the L1 caches, fueled by flexible lifetime choices. As seen in Figure 11, TCR++ shows remarkable improvement in L1 cache performance over TCR (with an average of 25.4% decrease in L1 miss rate, within 1.2% of MESI). Specifically, with the detailed read misses breakdown in Figure 11, we can see that the read misses due to lifetime expiration is decreased significantly. In most workloads (9 out of 15), the lifetime expiration induced read misses are barely noticeable. The small number of read misses on expired lines well reflects the efficiency of the proposed lifetime prediction mechanism. In particular, the SharedRO optimization contributes significantly to the improved L1 cache performance, as L1 hits on SharedRO state takes up a considerable part of L1 shared read hits in Figure 12.

The reduction in L1 miss rate translates to less network traffic. On an average, The network traffic of TCR++ is within 1.3% of the baseline MESI (with the best case reduction of 18.2% for `raytrace`) and 33.9% reduction over TCR. TCR++ shows similar network traffic compared to the baseline MESI directory protocol. TCR++ does not have invalidation traffic where a write needs to invalidate other shared copies as in a directory protocol. But as we maintain ownership in the L2, TCR++ still has the network traffic caused by ownership shift or downgrade requests. Besides, TCR++ also incurs network traffic due to self-invalidations and signature transfers.

Impact of infinite Bloom filter size: As shown in Figure 8, by varying the Bloom filter size from 256-bit to an idealized infinite size, TCR++Inf shows little difference in execution time and network traffic (both within 1%), compared to TCR++. In fact, as we can see in Figure 11, for TCR++ with a 256-bit filter implementation, the Bloom filter induced read misses are fairly small across all workloads. TCR++Inf removes read misses caused by Bloom filter false positive hits; however, the L1 miss rate reduction is minimal (0.5%), which does not translate to performance improvement. Thanks to the timestamp assigned to every signature that allows the signature to be cleared after its timestamp expiration, unnecessary L1 misses are saved. Overall, TCR++ with a realistic Bloom filter configuration performs nearly identical to an infinite size Bloom filter.

Speculation: Although we evaluate our coherence protocols with simple cores, they can also be used to advantage with complex cores that support speculation and dynamic execution. In this case, a new possibility for optimization opens up. When an L1 Shared line has expired its timestamp and the Timestamp-Bypass (TB) bit is not set, TC-Release++ needs to access the L2 for a new lease. With speculative

execution mechanisms available, the latency of these L1 misses can be hidden by speculatively using the expired data in the L1 and the protocol reloads expired cache lines with their latest version from the L2. If the self-invalidated data and the L2 data differ (i.e., the line has been modified), speculation is squashed. This optimization is similar to [31] [5]. In Figure 13, we show the performance improvement by speculation using gem5's in-order core model with speculation mechanisms implemented and `fluidanimate` which has the most expiration-induced L1 misses. We can see that speculation improves the performance of TCR++ from within 0.5% of MESI to outperform it by 3.2%.

7 Related Work

We have discussed in passing the closest works to our proposal. Here we discuss other related work and provide a broader overview of more scalable approaches to coherence.

7.1 Timestamp-based coherence

Using timestamps for cache coherence has been explored in software [32] [33], significant software support increases burden to programmer beyond simple synchronization annotation. Nandy et al. [34] first investigated the use of timestamps for hardware coherence, but important aspects like the target memory consistency model and lifetime prediction are not discussed. In addition to the timestamp-based hardware coherence protocols we have discussed [3] [4], Tardis [5] [35] is a recently proposed work that relies on timestamps for maintaining coherence. Different from our proposal, Tardis is implemented for stronger consistency models (i.e., Tardis [5] models SC and Tardis 2.0 [35] models TSO), and it uses logical time and the novel time travel mechanism to eliminate the stalls on writes. Nevertheless, its logical time use incurs drawbacks like livelocks and legion L1 timestamp renewal requests, the proposed solutions [35] increase complexity in the protocol. Tardis also proposes some valuable optimizations in timestamp-based coherence: the performance loss due to its large number of premature expirations of L1 lines is hidden by speculatively making use of the data stored in the expired lines. It also introduces a timestamp compression mechanism to reduce the storage requirement. These optimizations are orthogonal to our proposal.

Elver et al. [36] [37] also use timestamps in the proposed coherence protocol for relaxed memory consistency models, but different from the timestamps in our proposal that indicates the lifetime of an L1 line, the purpose of using timestamps in [36] [37] is to transitively reduce the number of self-invalidations at acquires.

7.2 Coherence for relaxed consistency

Dynamic Self-Invalidation (DSI) [38] first proposed self-invalidation of lines in private caches, reducing coherence traffic as invalidations are no longer sent from the directory. The authors observed that for relaxed memory consistency models, as long as private lines are eliminated before the next synchronization point, coherence is guaranteed. Cache coherence for relaxed memory consistency has been explored in more recent work [39] [36] [37] [40] [41] [**?**]. VIPS-M [39] investigates coherence simplification fueled by relaxed consistency semantics. With proper private/shared data classification and a write-through cache, it builds a coherence protocol that brings down the need of coherence states to two rudimentary valid/invalid states. Similarly, Denovo protocols [41] [42] [**?**] achieve comparable complexity-efficiency and performance with however, more software involvement. Callback [40] proposes an elegant approach to eliminating the spin-waiting races in coherence protocols based on self-invalidation.

TSO-CC [36] make a major step toward adopting self-invalidating coherence protocols for more practical consistency models (i.e., TSO), while previous work commonly assume a RC consistency model. RC3 [37] explores the intersection of stricter consistency models (such as x86-64) and the recent convergence toward the adoption of RC consistency model by high-level programming languages.

In contrast to our proposal that uses a signature to selectively self-invalidate L1 lines, these approaches apply cache-wide self-invalidations at acquires that may degrade performance. Specifically, we expect the implementation of TC-Release with fixed zero-cycle lifetime to perform similar to a simple relaxed consistency coherence protocol that invalidates all L1 Shared lines at acquires. As suggested by the resulting performance (∼10% slower than MESI), a lot of shared lines will be unnecessarily victimized due to cache-wide invalidation.

7.3 Using signatures in coherence

Signatures have been used in transactional memory systems as conflict detectors [43]. In the context of RC, the idea of using a signature to track a the write set of a core before a release has been proposed in prior works [44] [42]. A primary problem associated with the signature design is when to clear the signatures. Over the execution of the workloads, the write-set tracked in the signature will grow very large, ultimately causing it to be saturated, where every signature lookup will result in a false positive hit. Prior works rely on software or compiler to perform signature clear [44] [42].

The major difference of our proposal with prior work is that our signature design is built on top of timestamp-based coherence protocol, which establishes the validity period of the signature. The timestamp of a signature provides a time bound by which the filter field can be safely cleared. Therefore, the signature clearing in our proposal is entirely hardware driven and does not require any software involvement. Additionally, the timestamp coherence also opens up further optimization opportunity — the W-FIFO effectively reduces the write-set size because globally visible writes (i.e., ones with expired timestamps) from the W-FIFO are not required to be tracked in the signature.

8 Conclusion

In this paper, we propose a timestamp-based coherence protocol for release consistency memory models that addresses the scalability issues in efficiently supporting cache coherence in large-scale systems. Our protocol is inspired by a recently proposed timestamp-based coherence protocol targeting GPU architectures [4]. However, we observe that implementing a similar coherence protocol for general-purpose many-core architectures leads to sub-par performance compared to the de-facto standard directory coherence protocols. To overcome the limitations and overheads,

we propose TC-Release++ that eliminates the expensive memory stalls and provides an optimized lifetime prediction mechanism. Compared to a conventional directory coherence protocol, TC-Release++ is highly scalable as it eliminates the storage overhead for coherence substantially but at the same time exhibits better execution time and comparable network traffic.

REFERENCES

[1] M. M. Martin *et al.*, "Why On-Chip Cache Coherence is Here to Stay,"

[2] D. J. Sorin *et al.*, "A Primer on Memory Consistency and Cache Coherence," *Morgan and Claypool Publishers, 2011*.

[3] M. Lis *et al.*, "Memory Coherence in the Age of Multicores," in *ICCD, 2011*.

[4] I. Singh *et al.*, "Cache Coherence for GPU Architectures," in *HPCA, 2013*.

[5] X. Yu and S. Devadas, "Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory," in *PACT, 2015*.

[6] K. Gharachorloo *et al.*, "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors," *ISCA, 1990*.

[7] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *ISCA*, 1992.

[8] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC, 2009*.

[9] Z. Hu *et al.*, "Timekeeping in the memory system: predicting and optimizing memory behavior," in *ISCA*, 2002.

[10] J. Goodacre and A. N. Sloss, "Parallelism and the arm instruction set architecture," *IEEE Computer*, 2005.

[11] Compaq, "Alpha 21264 microprocessor hardware reference manual," 1999.

[12] "Formal specification of intel itanium processor family memory ordering," 2002.

[13] B. Hechtman *et al.*, "QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs," *HPCA, 2014*.

[14] S. V. Adve and M. D. Hill, "Weak orderinga new definition," in *ISCA*, 1990.

[15] D. L. Dill, "The murphi verification system," in *CAV*, 1996.

[16] X. Chen *et al.*, "Reducing verification complexity of a multicore coherence protocol using assume/guarantee," in *FMCAD*, 2006.

[17] K. L. McMillan, "Parameterized verification of the flash cache coherence protocol by compositional model checking," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2001.

[18] N. Binkert *et al.*, "The gem5 Simulator," *Computer Architecture News, 2011*.

[19] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi-core runtime," in *PPoPP*, 2006.

[20] H. Sung and S. V. Adve, "DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations," in *ASPLOS, 2015*.

[21] J. Alsop *et al.*, "Lazy release consistency for gpus," in *MICRO, 2016*, 2016.

[22] E. Clarke *et al.*, "Progress on the state explosion problem in model checking," in *Informatics*, 2001.

[23] T. Tsuchiya and A. Schiper, "Model checking of consensus algorithms," in *SRDS*, 2007.

[24] F. Derepas *et al.*, "Avoiding state explosion for distributed systems with timestamps," in *International Symposium of Formal Methods Europe*, 2001.

[25] G. Jakubowska *et al.*, "Verifying security protocols with timestamps via translation to timed automata," in *Proc. of the International Workshop on Concurrency, Specification and Programming (CS&P'05)*, 2005.

[26] C. Bienia *et al.*, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT, 2008*.

[27] S. C. Woo *et al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA, 1995*.

[28] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *HPCA, 2012*.

[29] M. Ferdman *et al.*, "Cuckoo Directory: A Scalable Directory for Many-Core Systems," in *HPCA, 2011*.

[30] D. Wendel *et al.*, "The Implementation of POWER7 TM: A Highly Parallel and Scalable Multi-Core High-End server Processor," in *ISSCC, 2010*.

[31] J. Huh *et al.*, "Coherence decoupling: Making use of incoherence," in *ASPLOS, 2004*.

[32] S. L. Min and J.-L. Baer, "Design and Analysis of A Scalable Cache Coherence Scheme Based on Clocks and Timestamps," *TPDS, 1992*.

[33] X. Yuan *et al.*, "A Timestamp-Based Selective Invalidation Scheme for Multiprocessor Cache Coherence," in *ICPP, 1996*.

[34] S. Nandy and R. Narayan, "An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems," in *International Workshop on Parallel Processing, 1994*.

[35] X. Yu *et al.*, "Tardis 2.0: Optimized time traveling coherence for relaxed consistency models," in *PACT*, 2016.

[36] M. Elver and V. Nagarajan, "TSO-CC: Consistency Directed Cache Coherence for TSO," *HPCA, 2014*.

[37] M. Elver and V. Nagarajan, "RC3: Consistency Directed Cache Coherence for x86-64 with RC Extensions," *PACT, 2015*.

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

[39] A. Ros and S. Kaxiras, "Complexity-Effective Multicore Coherence," *PACT, 2012*.

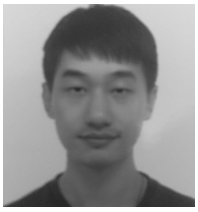
[40] A. Ros and S. Kaxiras, "Callback: Efficient Synchronization without Invalidation with A Directory Just for Spin-Waiting," *ISCA, 2015*.

[41] B. Choi *et al.*, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *PACT, 2011*.

[42] H. Sung *et al.*, "DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism," in *ASPLOS, 2013*.

[43] L. Ceze *et al.*, "Bulk disambiguation of speculative threads in multiprocessors," in *ISCA*, 2006.

[44] T. J. Ashby *et al.*, "Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters," *IEEE Transactions on Computers, 2011*.



Yuan Yao received the BS degree in computer science from Zhejiang University, Hangzhou, China. He is currently working toward the PhD degree at the School of Computer Science, Zhejiang University. His current research interests include cache coherence protocols and memory hierarchies. He is a student member of the IEEE and the ACM.

Wenzhi Chen received the BS, MS, and PhD degrees in computer science and technology from Zhejiang University, Hangzhou, China. He is currently a professor at the School of Computer Science and Technology at Zhejiang University. His areas of research include computer architecture, system software, embedded system, and network security. He is a member of the IEEE and the ACM.

Tulika Mitra received the PhD degree in computer science from the State University of New York at Stony Brook, Stony Brook, NY, USA. She is a Professor of computer science with the School of Computing, National University of Singapore, Singapore. She has authored more than 100 scientific publications. Her research interests include embedded real-time systems, energy-efficient computing, and heterogeneous computing systems.

Yang Xiang received the PhD degree in computer science from Deakin University, Australia. He is currently a full professor at School of Information Technology, Deakin University. He is the director of the Network Security and Computing Lab (NSCLab). His research interests include network and system security, distributed systems, and networking. He is a senior member of the IEEE.