Efficient Timestamp-Based Cache Coherence Protocol for Many-Core Architectures

Yuan Yao[1], Guanhua Wang[2], Zhiguo Ge[3], Tulika Mitra[2], Wenzhi Chen[1] and Naxin Zhang[3]
[1]College of Computer Science and Technology, Zhejiang University
[2]School of Computing, National University of Singapore
[3]Huawei Singapore Research Centre
Email: yuanyao@zju.edu.cn, guanhua@comp.nus.edu.sg, ge.zhiguo@huawei.com, tulika@comp.nus.edu.sg, chenwz@zju.edu.cn, naxin.zhang@huawei.com

ABSTRACT

As we enter the era of many-core, providing the shared memory abstraction through cache coherence has become progressively difficult. The de-facto standard directory-based cache coherence has been extensively studied; but it does not scale well with increasing core count. Timestamp-based hardware coherence protocols introduced recently offer an attractive alternative solution. In this paper, we propose a timestamp-based coherence protocol, called *TC-Release++*, that addresses the scalability issues of efficiently supporting 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* coherence 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 coherence protocols. We overcome the limitations and overheads of TC-Release by introducing 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 (overhead for coherence per last-level cache line scales logarithmically with core count as opposed to linearly for directory coherence) and shows better execution time (3.0%) and comparable network traffic (within 1.3%) relative to the baseline MESI directory coherence protocol.

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.

Significant effort has been invested to make the directory coherence more scalable by exploiting efficient sharer-tracking representation [3][4][5][6][7], hierarchical directories [8][9], and eliminating directory for private data [10]. Other approaches like [9][11] investigate better directory organization and management policy to achieve more efficient utilization.

An alternative approach to directory coherence are the recently proposed timestamp-based coherence protocols [12][13][14][15] that remove the scalability burden associated with directory coherence. In directory coherence, the directory maintains information about all the private caches that share a memory line. On a write to a cache line, the directory sends out explicit invalidation requests to all the sharers of the cache line and waits for the acknowledgments. Thus after a write, there is only one cache line with the valid data. 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) [12][13] — a timestamp coherence protocol — maintains coherence by stalling a write at the L2 cache controller until the timestamps of all the L1 private cache copies have expired and thus they have been self-invalidated. This write stall is necessary for Sequential Consistency (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

ICS '16, June 01-03, 2016, Istanbul, Turkey
© 2016 ACM. ISBN 978-1-4503-4361-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926270

consistency models relax some of the ordering requirements. For example, Release Consistency (RC) model [16] 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. As most modern processors adopt relaxed memory consistency models, and with the recent adoption of RC in high-level programming languages like C++11 and Java [2], building a highly scalable coherence protocol by exploiting RC memory model is desirable and worthwhile.

TC-Weak [14] 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 (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++ that adopts simple hardware support to eliminate the significant memory stalls involved in TC-Release, and an optimized lifetime prediction mechanism to improve cache performance. The resulting coherence protocol has storage requirement for timestamps per cache line that 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.

The remainder of this paper is organized as follows. Section 2 describes the design of TC-Release. Section 3 details the improved design TC-Release++. Section 4 presents the methodology we use for experiments. Section 5 provides the evaluation results. Section 6 discusses related work. Section 8 concludes this work.

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 [14] 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) [12][13] 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 is based on the insight 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.

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 [14], 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. Figure 1 shows the breakdown of L1 reads in the baseline MESI directory protocol for 15 multi-threaded workloads. Simulation details can be found in Section 4. 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 [17]. 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 2 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 (where the area and energy efficiency is consistent with the prevailing trend towards many-core scaling [18][19]) and only one GWCT is maintained per L1 cache.

Figure 3 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

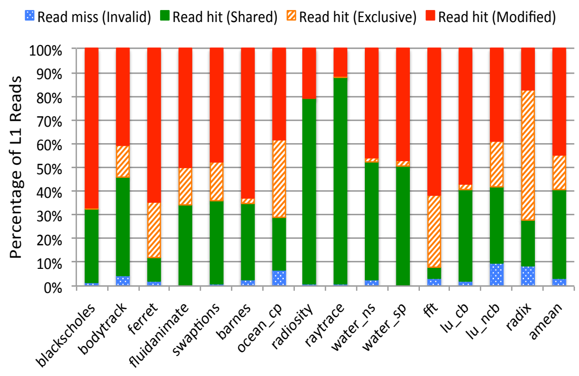


Figure 1: L1 read hits and misses in the baseline MESI directory protocol, with breakdown of reads in different states.

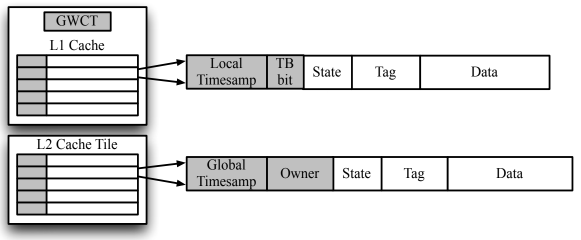


Figure 2: Hardware extensions for TC-Release. A GWCT is added per L1 cache. Each L1 and L2 line is extended with a timestamp. The owner field in an L2 line tracks the exclusive owner. The use of Timestamp Bypass (TB) bit added per L1 line is for the RC-optimization, detailed in Section 2.2.

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) 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

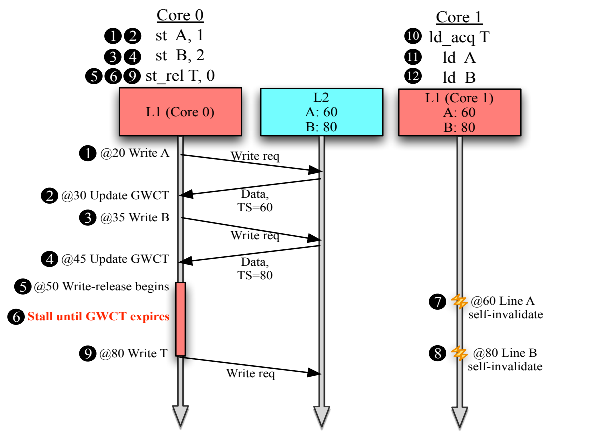


Figure 3: A simplified example of TC-Release with the execution of the code segment shown at the top.

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 2). 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 ownership. 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 4, 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. Before performing another acquire, Core 1 executes 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
```

Figure 4: 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 2. 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 Acquire $\rightarrow$ Reads/Writes ordering, all the TB bits are reset after a read-acquire operation.

2.3 TC-Release vs. TC-Weak

TC-Release and TC-Weak both implement Release Consistency. Similar to TC-Weak, which stalls on memory fence instructions, TC-Release stalls on a write-release operation. This stalling guarantees that the memory locations modified by the core reach a coherent state before a release, with respect to the other cores that want to communicate with it.

In contrast to using write-through caches in TC-Weak, TC-Releases uses write-back caches. A write-through cache exploits streaming memory access commonly found in GPU workloads. But it will not perform as efficiently for general-purpose CPU workloads that exhibit significantly higher temporal locality (on an average 96.9% of the reads hit in the L1, as seen in Figure 1).

Taking advantage of a write-back cache, TC-Release further decouples L1 private lines (lines in Exclusive/Modified)

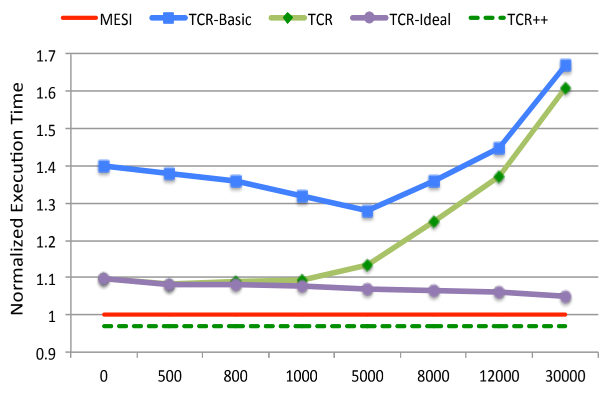


Figure 5: Normalized execution time of TCR, TCR-Basic, TCR-Ideal with various fixed lifetimes, with respect to baseline MESI directory protocol and TCR++.

from the timestamp-based coherence, as L1 lines in only Shared state can incur timestamp expiration. By tracking the exclusive ownership in the L2 and allowing the L1 to indefinitely cache exclusive data, TC-Release eliminates the L1 misses caused by self-invalidations of private data. As shown in Figure 1, large portion of reads are to L1 Exclusive/Modified lines (59.1% on an average), which are free from timestamp expiration induced self-invalidations in TC-Release.

TC-Release also implements the RC-optimization that avoids redundant self-invalidations for L1 lines. For one acquire, an L1 line can be self-invalidated up to once. In contrast, TC-Weak can potentially self-invalidate expired lines multiple times per acquire. RC-optimization significantly improves performance, as will be outlined in the following subsection and in Section 5 with detailed performance results.

2.4 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 5 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 5, 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 5, 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 5, 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 5, 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 than the baseline MESI directory protocol (plotted in dashed green line in Figure 5). 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

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 re-

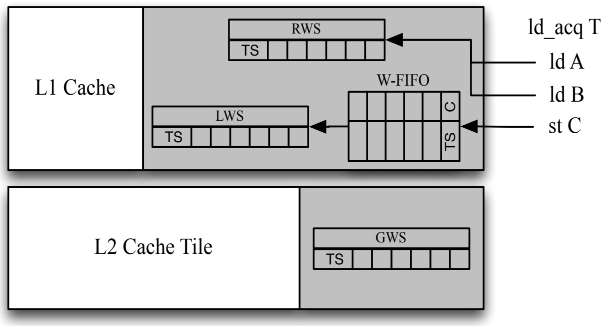


Figure 6: Hardware extensions for the signature design. In the given code segment on the right, initially A and B are located in two L1 Shared lines. The write to address C returns a timestamp.

lease. 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 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.

3.1.1 Hardware extensions and protocol design

Figure 6 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.

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 6. 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 6, 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 6, 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.

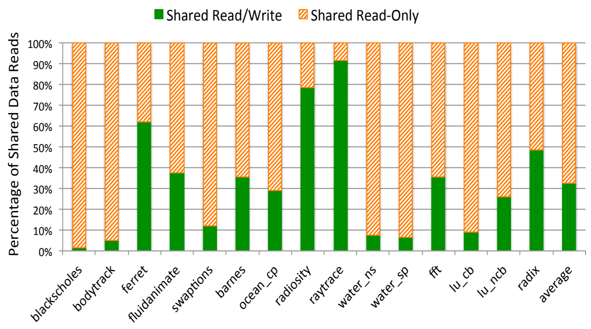
3.1.2 Comparison with related works using signature

In the context of RC, the idea of using a signature to track a the write set of a core before a release and conveying it to the next core that performs the corresponding acquire has been proposed in prior works [21][22]. 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. In the worst case, every signature lookup will result in a false positive hit, resulting in unnecessary cache line self-invalidations. Prior works rely on software or compiler to perform signature clear [21][22]. Ashby et al. [21] extensively modify the barrier primitive to make sure that all the writes have become globally visible before the barrier is released. They propose to insert a special Bloom filter reset instruction at the end of the barrier primitive that informs the hardware to clear the Bloom filter. As barrier primitives are infrequently invoked in parallel workloads [23][24], the signatures can still easily get saturated, driving the false positive rate very high. DeNovoND [22] requires heavy involvement from the application programmer to provide the information regarding parallel program phase boundaries and the read/write effects of the memory regions manipulated in each phase. With explicit software annotations, it clears all the signatures after a parallel phase ends or when a barrier primitive completes. DeNovoND also proposes a signature clearing mechanism that requires cache-wide self-invalidation of all potentially stale data incurring additional overheads.

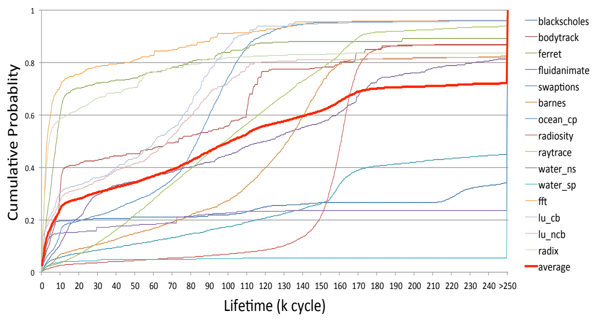
The major difference of our proposal with prior work is that our signature design is built on top of timestamp-based coherence protocol, which establishes the validity period of the signature. The timestamp of a signature provides a time bound by which the filter field can be safely cleared. Therefore, the signature clearing in our proposal is entirely hardware driven and does not require any programmer and/or compiler 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.

3.2 Lifetime Prediction and Shared Read-Only Optimization

3.2.1 Workload characterizations



(a) Breakdown of shared data read accesses into accesses to shared read-only and shared read-write lines.



(b) Cumulative probability distribution of the re-use of L1 shared read/write lines with respect to the lifetime of the lines.

Figure 7: Workload characterization

We first perform workload characterizations to motivate our lifetime prediction mechanism in TCR++.

Shared read-only lines: Figure 7a shows the breakdown of all the shared data reads in the baseline MESI protocol into shared read/write and shared read-only lines. A considerable fraction (68% on an average) of shared reads access read-only lines. The read-only lines do not suffer from coherency issues. Therefore, in our timestamp-based coherence, shared read-only lines should stay in the L1 caches as long as possible similar to exclusive cache lines; these lines do not need timestamps and consequently do not need lifetime prediction.

Lifetime and re-usability of shared read/write lines: We also explore the re-usability of shared read/write lines with respect to their lifetime. We first define the lifetime TL of an L1 shared read/write line k in the baseline directory protocol as the time from it transition to Shared state in the L1 cache (TR_k) until it gets invalidated or upgraded as a result of a write to the line (TW_k).

$$TL_k = TW_k - TR_k$$

We define N_k as the number of re-uses throughout the lifetime of an L1 shared line k and d_t is the sum of L1 re-uses of all the lines with a lifetime of t:

$$d_t = \sum_{TL_k = t} N_k$$

Assume MAX is the largest lifetime found among all L1 cache lines. The total number of re-uses of all L1 shared read/write lines $\mathcal{M}$ can be denoted as:

$$\mathcal{M} = \sum_{t=0}^{MAX} d_t$$

We define ρ_t as the ratio of the re-uses of shared L1 cache lines with a lifetime t to the total re-use of all L1 shared read/write lines:

$$\rho_t = d_t / \mathcal{M}$$

Probability Distribution of ρ: To obtain the probability distribution of ρ, we modify the baseline MESI directory protocol to record the start and end of the lifetime of each memory location. This is achieved by making L1 cache size large enough to eliminate the perturbation due to L1 evictions.

Figure 7b plots the cumulative probability distribution of ρ for all workloads. The X-axis is the lifetime and the Y-axis is the cumulative probability $\rho_{t \leq x}$. The red bold curve is the average cumulative probability distribution of all 15 workloads. Figure 7b reveals huge variability in the re-use of L1 shared read/write lines both within and across applications.

For most workloads, large lifetimes is preferable, as over 50% (on an average) of the re-uses belong to cache lines with lifetime greater than 100K cycles. However, a notable fraction of re-uses of L1 shared read/write lines are to lines with short- or medium-sized lifetimes. For example, lines with lifetime less than 5K cycles cover more than 50% of re-uses for `fft` and `radix`, because of frequent writes to shared data in these workloads. On an average, 25% of all re-uses fall to cache lines with less than 10K cycles lifetime. Note that `water_nsqured, water_spatial, blackscholes` and `swaptions` demonstrate significant re-use of cache lines with extremely long lifetimes (longer than 250K cycles) because their shared read/write lines are very infrequently written.

Overall, in order to cater to the differing lifetime preferences across cache lines, the lifetime predictor should be able to swiftly adjust lifetime values, and one single lifetime value for all accesses as proposed in TC-Weak [14] may be inadequate.

3.2.2 Lifetime prediction for access patterns

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.

4. METHODOLOGY

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

4.1 Simulation Environment

For evaluation of our proposal, we use the *gem5* full-system simulator [25] with Ruby memory system enabled. A 64-tile 2D mesh network-on-chip is modeled by Garnet [26]. Table 1 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 4.2 for details). We use the H3 Bloom filter implementation, with four hashing functions and a 256-bit filter. 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, as none of the workloads trigger a timestamp rollover. Timestamp rollover solution has been discussed in [14] and further exploration is reserved for future work.

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 sharer vector (i.e., 64-bit in our case) is stored in every LLC entry to precisely track the sharers.

Table 1: Simulation parameters.

Cores	64 in-order cores at 2 GHz, Alpha ISA, single-thread
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 2: 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

4.2 Workloads

We use PARSEC [23] and SPLASH-2 [24] workloads to evaluate our proposal. Table 2 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.

5. 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

Table 3: 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

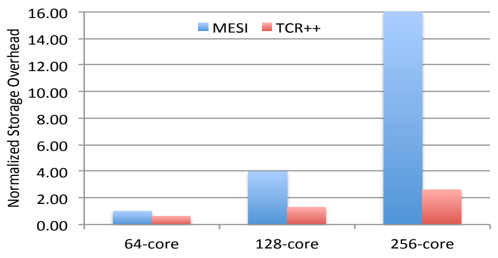


Figure 8: Storage overheads for cache coherence in TCR++ and MESI, with up to 256 cores.

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 [14] 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 (refer to the average curve in Figure 7b). 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.

5.1 Storage overheads

Table 3 shows the storage requirements for TCR++. The per line storage requirement for maintaining the timestamp has the most significant impact on hardware cost. The addi-

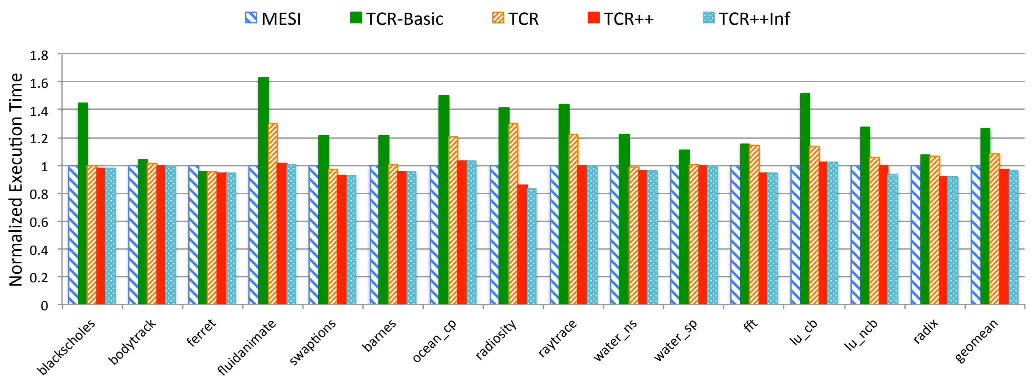


Figure 9: Execution time of all configurations, normalized to MESI.

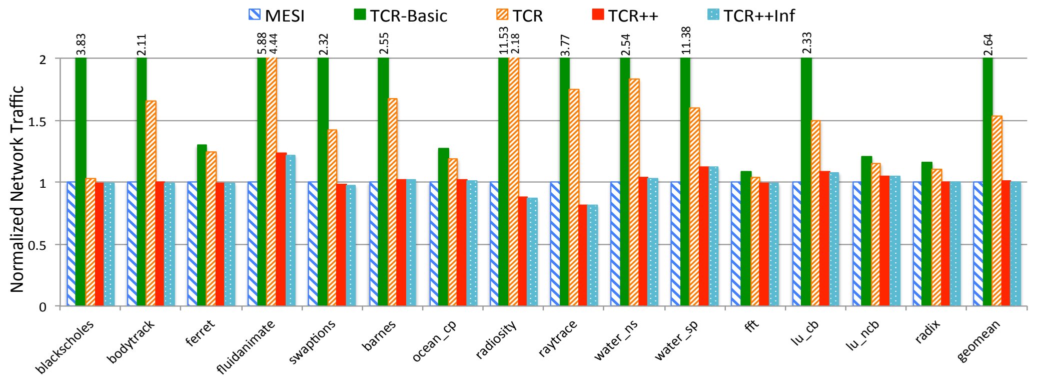


Figure 10: Network traffic of all configurations, normalized to MESI.

tional storage overheads for enabling 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.

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 8 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.

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 [27][28][29]). As illustrated in Figure 8, TCR++ reduces the directory storage overhead by 83% compared to MESI at 256-core, which can directly translate to significant savings in chip area.

5.2 Performance results

Figure 9 and Figure 10 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 10, TCR has an average increase of 53.2% in network traffic over MESI.

TCR shows worse performance for workloads with lots of

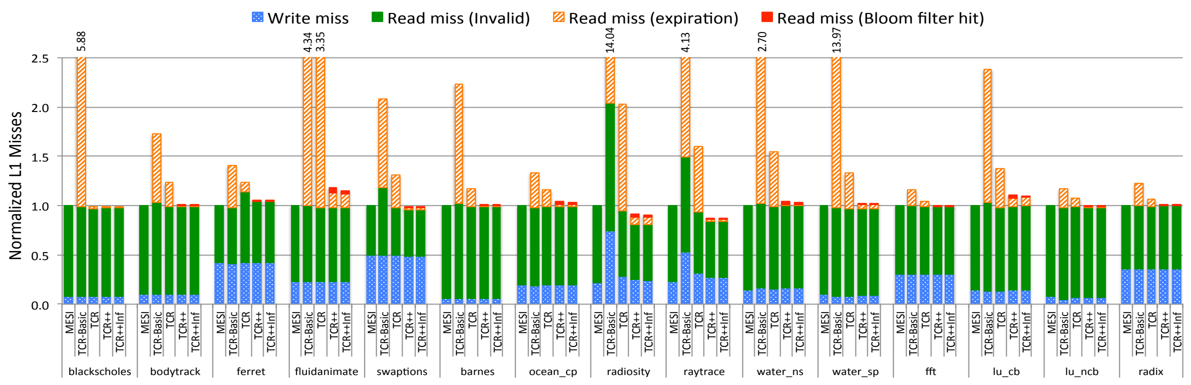


Figure 11: L1 miss rate of evaluated configurations, normalized to the baseline MESI. Misses are broken down by writes and reads, with the latter split up by three causes: Invalid state, lifetime expiration, and Bloom filter hit.

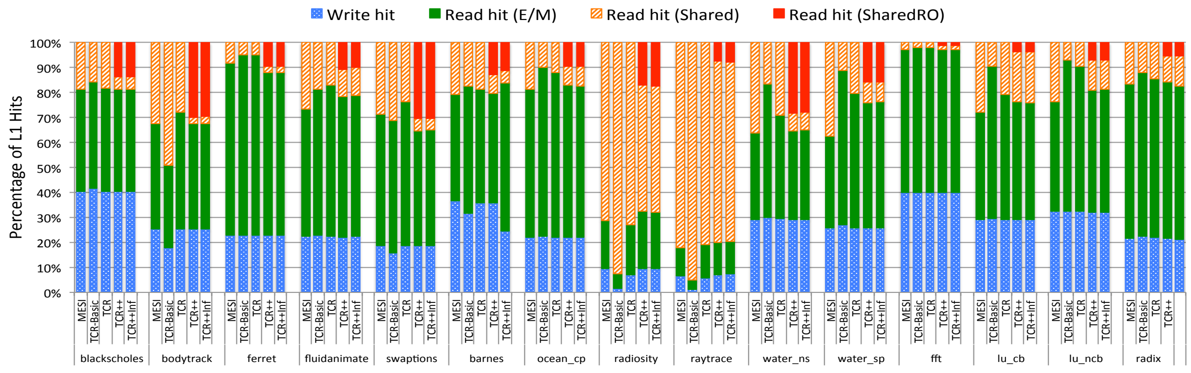


Figure 12: L1 hits breakdown by writes and reads, with the latter split up by cache states: Exclusive/Modified, Shared and SharedRO.

shared data accesses and frequent synchronizations. It reveals mediocre performance for workloads with small shared data working set and predominant accesses to private 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 9, 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.

6. RELATED WORK

We have discussed in passing the closest works to our proposal. Here we discuss the other related works.

Using timestamps for cache coherence has been explored in software [30][31]. Nandy et al. [32] first investigated the use of timestamps for hardware coherence. In addition to the timestamp-based hardware coherence protocols we have discussed [12][13][14], Tardis [15] is a recently proposed work that relies on timestamps for maintaining coherence. Different from our proposal, Tardis is implemented for Sequential Consistency, and it uses logical time and the novel time travel mechanism to eliminate the stall on writes. Besides, it 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. [33][34] 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 [33][34] is to transitively reduce the number of self-invalidations at acquires.

Dynamic Self-Invalidation (DSI) [35] 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 [20][33][34][36][37][38][39]. 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. 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 [14]. 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.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This work was supported by Huawei International Pte. Ltd. research grant.

9. REFERENCES

[1] M. M. Martin, M. D. Hill, and D. J. Sorin, "Why On-Chip Cache Coherence is Here to Stay," *Communications of the ACM, 2012.*

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

[3] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes.," in *International Conference for Parallel Processing, 1990.*

[4] Z. Hongzhou, A. Shriraman, and S. Dwarkadas, "SPACE: Sharing Pattern-Based Directory Coherence for Multicore Scalability," in *International Conference on Parallel Architectures and Compilation Techniques, 2010.*

[5] M. Alisafaee, "Spatiotemporal Coherence Tracking," in *International Symposium on Microarchitecture, 2012.*

[6] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-Grain Coherence Directories," in *International Symposium on Microarchitecture, 2013.*

[7] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang, "SelectDirectory: A Selective Directory for Cache Coherence in Many-Core Architectures," in *Design, Automation and Test in Europe, 2015.*

[8] L. Zhang, D. Strukov, H. Saadeldeen, D. Fan, M. Zhang, and D. Franklin, "SpongeDirectory: Flexible Sparse Directories Utilizing Multi-Level Memristors," in *International Conference on Parallel Architectures and Compilation Techniques, 2014.*

[9] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *International Symposium on High-Performance Computer Architecture, 2012.*

[10] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks," in *International Symposium on Computer Architecture, 2011.*

[11] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo Directory: A Scalable Directory for Many-Core Systems," in *International Symposium on High-Performance Computer Architecture, 2011.*

[12] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas, "Memory Coherence in the Age of Multicores," in *International Conference on Computer Design, 2011.*

[13] K. S. Shim, M. H. Cho, M. Lis, and S. Devadas, “Library Cache Coherence,” in *Csail technical report, 2011.*

[14] I. Singh, A. Shriraman, W. W. Fung, M. O’Connor, and T. M. Aamodt, “Cache Coherence for GPU Architectures,” in *International Symposium on High-Performance Computer Architecture, 2013.*

[15] X. Yu and S. Devadas, “Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory,” in *International Conference on Parallel Architectures and Compilation Techniques, 2015.*

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, “Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors,” *International Symposium on Computer Architecture, 1990.*

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A Benchmark Suite for Heterogeneous Computing,” in *International Symposium on Workload Characterization, 2009.*

[18] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, “Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing,” *International Symposium on Computer Architecture, 2011.*

[19] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, “The Case for A Single-Chip Multiprocessor,” *International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.*

[20] A. Ros and S. Kaxiras, “Complexity-Effective Multicore Coherence,” *International Conference on Parallel Architectures and Compilation Techniques, 2012.*

[21] T. J. Ashby, P. Diaz, and M. Cintra, “Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters,” *IEEE Transactions on Computers, 2011.*

[22] H. Sung, R. Komuravelli, and S. V. Adve, “DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism,” in *International Conference on Architectural Support for Programming Languages and Operating Systems, 2013.*

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *International Conference on Parallel Architectures and Compilation Techniques, 2008.*

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

[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, “The gem5 Simulator,” *Computer Architecture News, 2011.*

[26] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, “GARNET: A Detailed On-Chip Network Model inside A Full-System Simulator,” in *International Symposium on Performance Analysis of Systems and Software, 2009.*

[27] D. Wendel, R. Kalla, R. Cargoni, J. Clables, J. Friedrich, R. Frech, J. Kahle, B. Sinharoy, W. Starke, S. Taylor, S. Weitzel, S. G. Chu, S. Islam, and V. Zyuban, “The Implementation of POWER7 TM: A Highly Parallel and Scalable Multi-Core High-End server Processor,” in *International Solid-State Circuits Conference, 2010.*

[28] A. Basu, D. R. Hower, M. D. Hill, and M. M. Swift, “Freshcache: Statically and Dynamically Exploiting Dataless Ways,” in *International Conference on Computer Design, 2013.*

[29] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, “Reducing Cache Power with Low-Cost, Multi-Bit Error-Correcting Codes,” *International Symposium on Computer Architecture, 2010.*

[30] S. L. Min and J.-L. Baer, “Design and Analysis of A Scalable Cache Coherence Scheme Based on Clocks and Timestamps,” *IEEE Transactions on Parallel and Distributed Systems, 1992.*

[31] X. Yuan, R. Melhem, and R. Gupta, “A Timestamp-Based Selective Invalidation Scheme for Multiprocessor Cache Coherence,” in *International Conference for Parallel Processing, 1996.*

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

[33] M. Elver and V. Nagarajan, “TSO-CC: Consistency Directed Cache Coherence for TSO,” *International Symposium on High-Performance Computer Architecture, 2014.*

[34] M. Elver and V. Nagarajan, “RC3: Consistency Directed Cache Coherence for x86-64 with RC Extensions,” *International Conference on Parallel Architectures and Compilation Techniques, 2015.*

[35] A. R. Lebeck and D. A. Wood, “Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors,” in *International Symposium on Computer Architecture, 1995.*

[36] S. Kaxiras and G. Keramidas, “SARC Coherence: Scaling Directory Cache Coherence in Performance and Power,” *IEEE Micro, 2010.*

[37] A. Ros and S. Kaxiras, “Callback: Efficient Synchronization without Invalidation with A Directory Just for Spin-Waiting,” *International Symposium on Computer Architecture, 2015.*

[38] 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 *International Conference on Parallel Architectures and Compilation Techniques, 2011.*

[39] H. Sung and S. V. Adve, “DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations,” in *International Conference on Architectural Support for Programming Languages and Operating Systems, 2015.*