RC3: Consistency directed cache coherence for x86-64 with RC extensions

RC3: Consistency directed cache coherence for x86-64 with RC extensions

Marco Elver, Vijay Nagarajan
University of Edinburgh
{marco.elver, vijay.nagarajan}@ed.ac.uk

***Abstract*—The recent convergence towards programming language based memory consistency models has sparked renewed interest in lazy cache coherence protocols. These protocols exploit synchronization information by enforcing coherence only at synchronization boundaries via self-invalidation. In effect, such protocols do not require sharer tracking which benefits scalability. On the downside, such protocols are only readily applicable to a restricted set of consistency models, such as Release Consistency (RC), which expose synchronization information explicitly. In particular, existing architectures with stricter consistency models (such as x86-64) cannot readily make use of lazy coherence protocols without either: changing the architecture's consistency model to (a variant of) RC at the expense of backwards compatibility; or adapting the protocol to satisfy the stricter consistency model, thereby failing to benefit from synchronization information.**

We show an approach for the x86-64 architecture, which is a compromise between the two. First, we propose a mechanism to convey synchronization information via a simple ISA extension, while retaining backwards compatibility with legacy codes and older microarchitectures. Second, we propose RC3, a scalable hardware cache coherence protocol for RCtso, the resulting memory consistency model. RC3 does not track sharers, and relies on self-invalidation on acquires. To satisfy RCtso efficiently, the protocol reduces self-invalidations transitively using per-L1 timestamps only. RC3 outperforms a conventional lazy RC protocol by 12%, achieving performance comparable to a MESI directory protocol for RC optimized programs. RC3's storage overhead per cache line scales logarithmically with increasing core count, and reduces on-chip coherence storage overheads by 45% compared to a related approach specifically targeting TSO.

***Keywords*-multiprocessors; cache coherence; memory consistency models;**

I. Introduction

In recent years we have seen widespread convergence towards clearly defined programming language level memory consistency models, such as C11 [1], C++11 [2], [3] and Java [4]. These programmer-centric models require the programmer to explicitly distinguish and label data and synchronization operations at a much higher level of abstraction, rather than having to deal with the low level details of the hardware level consistency models [5]. It is beneficial to convey the language level labels to the hardware, as hardware can exploit this information for improved performance. Since data operations need not be ordered among themselves, there are fewer restrictions on, e.g. out-of-order pipeline implementations.

In addition to the performance benefits, cache coherence protocol implementations in multiprocessor systems can also exploit synchronization information, leading to more scalable protocols. Indeed, with synchronization operations exposed, coherence need only be enforced *lazily* at synchronization boundaries via self-invalidation [6], [7], [8], [9]. Using self-invalidation, instead of relying on *eager* invalidations, is beneficial, as it no longer requires maintaining a sharing vector and associated data structures for maintaining the list of sharers. Although there have been numerous approaches to optimize the cache and directory organization to maintain the list of sharers more efficiently [10], [11], [12], [13], [14], [15], for coherence protocols that exploit synchronization information, the sharing vector can be completely eliminated.

Motivation: Conveying data/synchronization information from the language level to the hardware level, however, requires a compatible hardware memory consistency model that also clearly distinguishes between data and synchronization operations. One such model, enabling an efficient mapping from the language to the hardware level, is Release Consistency (RC) [16]. In fact, a number of recent lazy coherence protocols [17], [18], [19], [20], [21], [22] target variants of RC.

Unfortunately, some existing architectures such as x86 only support stricter memory consistency models, e.g. x86-TSO [23], which cannot directly exploit the explicit data/synchronization information available at the language level. As there exists a well established ecosystem of software around these architectures, moving to a weaker RC variant is not an option as legacy code must continue to work. Therefore, most lazy coherence protocols cannot be applied to these architectures. One exception to this is the recently proposed TSO-CC protocol [24], which implements a lazy coherence protocol for Total Store Order (TSO). Although TSO-CC enables lazy coherence for x86 systems, there is still no way to exploit the synchronization information available at the language level.

Therefore, our research question is the following: how can architectures (such as x86) benefit from the explicit synchronization information available from language level memory consistency models? At the same time, legacy code which assumes the original hardware memory model (x86-TSO), must continue to work. We attack this problem for the widely deployed x86-64 architecture.

Approach: In x86-TSO, reads and writes already provide

acquire and release semantics respectively. Therefore, instead of adding additional acquire/release instructions to the ISA, we propose adding *ordinary* (relaxed) reads and writes to represent data operations (§III). This is realized via unused (null) prefixes which have become available in x86-64; the semantics of one unused prefix is changed to denote ordinary memory operations. The reads and writes from older legacy codes (that are not labeled with the extension) simply cause fewer instruction reorderings as the reads and writes are treated as acquires and releases, just as is the case in x86-TSO. The resulting memory consistency model is *RCtso* (x86-RCtso). While variants of RC, such as RCsc and RCpc can be found in the literature [16], RCtso is not explicitly mentioned. RCtso is similar to RCpc in that it relaxes the $w_R \rightarrow r_A$ [1] ordering, but unlike RCpc, requires multi-copy atomicity among synchronization operations.

To take advantage of RCtso, we propose the *RC3* coherence protocol (§IV): a lazy cache coherence protocol that targets RCtso. We base RC3 on the recently proposed TSO-CC protocol, as it provides an efficient lazy coherence protocol implementation for TSO. In RC3, however, we additionally exploit the exposed ordinary/synchronization information to optimize the protocol. In TSO, since synchronization information is unavailable, every read can potentially be an acquire. TSO-CC employs transitive reduction using timestamps to limit self-invalidation: upon a L1 miss, self-invalidation is only performed if the response's timestamp is larger than the last-seen timestamp of the writer. There is however a significant cost to performing this optimization: to achieve good performance, each cache line in both L1s and L2[2], needs to hold a timestamp due to absence of explicit synchronization information. In RC3, with data and synchronization information directly available, we no longer need to self-invalidate on ordinary reads. We observe, however, that there are performance gains to be realized when applying a limited form of transitive reduction optimization only to synchronization accesses, thereby reducing self-invalidations on redundant acquires, compared to a conventional RC lazy coherence protocol. Since synchronization accesses are relatively infrequent, we can perform this limited form of transitive reduction with *only per-L1 timestamps*, eliminating per cache line timestamps in both L1s and L2.

Contributions: Our key contribution is *RC3*, a lazy cache coherence protocol for *RCtso*, and a seamless approach to adopt the protocol in the x86-64 architecture – thereby allowing the architecture to exploit the explicit synchronization information present in many recent language level memory consistency models. We achieve this by showing how to convey explicit ordinary and synchronization information to the hardware via an ISA extension, and in doing so propose to change the consistency model from x86-TSO to x86-RCtso. The RC3 protocol then targets the RCtso consistency model lazily, without the need for a sharing vector nor per cache line timestamps.

In comparison to a conventional lazy RC coherence protocol, RC3 achieves a 12% performance improvement on average owing to transitive reduction of redundant acquires using timestamps. In comparison to TSO-CC, RC3 reduces coherence storage requirements by 45% by eliminating per cache line L1 and L2 timestamps. Furthermore, eliminating per cache line timestamps also simplifies cache accesses as timestamps do not need to be tagged.

II. Background

This section provides an overview of various approaches to memory consistency, and how the choice of the memory consistency model impacts the cache coherence protocol. For a more detailed discussion, we refer the reader to [16], [25], [26], [27]. This is followed by an overview of the recently proposed TSO-CC protocol (§II-B), which we base our protocol on.

A. Approaches to memory consistency models

While there are various options for deciding upon a memory consistency model in a multiprocessor system, it is essential to find the right balance between programmability and performance.

System-centric approach: In the system-centric approach, the memory consistency model is the direct interface with the hardware. In more relaxed consistency models, it becomes more difficult for programmers to reason about parallel programs, and as such, stricter models are preferred when programmers are expected to reason at the hardware level. Memory consistency models such as Sequential Consistency (SC) [28] and Total Store Order (TSO) [23] make it intuitive for a programmer to reason about parallel programs.

Unfortunately, implementations for these stricter consistency models typically result in fewer allowable optimizations by the hardware, and in the context of cache coherence, require *eager* coherence protocols. In eager coherence protocols, writes are propagated eagerly by invalidating or updating shared data in other caches [9].

Programmer-centric approach: While many commercial multiprocessor systems adopt very relaxed memory consistency models, giving architects fewer restrictions on optimizations, this usually complicates reasoning about parallel programs at the hardware level. This problem, however, can be solved if we assume that the programmer does not need to reason about programs using the system-centric consistency models, and instead is exposed to a higher level abstraction at the programming language level [5].

The only requirement of the hardware level consistency model then is that, any language level consistency model can

[1] r_A denotes a read acquire, w_R a write release; r, w and m denote ordinary (relaxed) reads, writes or any ordinary operations respectively; $\rightarrow$ is the happens before ordering relation between memory operations.

[2] We assume a local L1 cache per core and a NUCA architecture for the shared L2 cache.

be mapped to the hardware level. The formal basis for this approach can be found in Adve et al.'s *data-race-free* [25], [26], [29] and Gharachorloo et al.'s *properly labelled* [27], [16] models. In essence, the programmer explicitly labels synchronization and data operations correctly; in return the system (compiler and hardware) guarantees SC.

Modern programming languages are converging towards clearly defined memory consistency models, and as such, the programmer only needs to reason in terms of the language level consistency model. For instance, C++11 is an adaptation of *data-race-free-0* [3]. However, for hardware to be able to benefit from the explicit synchronization information, the hardware's consistency model should be able to distinguish between synchronization and data operations. A straightforward implementation of *data-race-free-0* is using RC [16] (without nsync) [25], where data operations are mapped to ordinary loads (r) and stores (w), and synchronization operations are mapped to acquires (r_A) and releases (w_R). RC requires maintaining $m \rightarrow w_R$ and $r_A \rightarrow m$, and depending on the RC-variant imposes restrictions on ordering between synchronization, e.g. RCsc requires that all possible orderings between synchronization are maintained.

As a result, the hardware benefits from additional opportunity for optimization, and in particular, coherence protocol implementations can be *lazy*. Propagation of ordinary memory operations can be delayed until an order can be re-established at synchronization boundaries [6], [7], [9]. This permits the protocol to remove the costly data structures to maintain a list of sharers, i.e. the sharing vector, and instead rely on self-invalidation upon synchronization boundaries as demonstrated by numerous prior works [7], [18], [21], [22], [30].

B. TSO-CC: Lazy coherence for TSO

Memory consistency models such as TSO, however, do not explicitly expose synchronization operations. In TSO, regular reads and writes have acquire and release semantics, respectively. *Then, at what point should a lazy coherence protocol self-invalidate?* Naïvely assuming every read or write to be synchronization can cause significant performance degradation. Despite this limitation, by exploiting the fact that TSO relaxes the $w \rightarrow r$ ordering, TSO does give rise to an efficient self-invalidation based lazy coherence protocol (without sharer tracking), as demonstrated by the recent TSO-CC protocol [24].

TSO-CC's first insight is that it is legal for a read to return a stale (locally cached) value, as long as the following hold. ① Periodic reads to a location eventually return the up-to-date copy of the value; TSO-CC accomplishes this by forcing a miss after a fixed number of hits – this effectively ensures the write propagation requirement of TSO. ② TSO's ordering requirements are not violated even though stale accesses are permitted; this is achieved by treating read misses (i.e. upon returning an up-to-date value) as acquires, which are followed by self-invalidation – this effectively ensures that the $r \rightarrow r$ ordering requirement is not violated. All other remaining ordering requirements ($m \rightarrow w$) are satisfied by propagating writes to the shared cache in order, and committing writes only after reads.

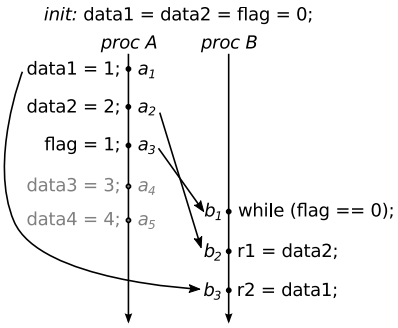

Figure 1. Message-passing example.

Considering every such read miss to be an acquire, however, causes excessive self-invalidations and degrades performance. *Which reads should be treated as acquires?* The second important insight concerns how to reduce excessive self-invalidations in the absence of explicit synchronization. To avoid redundant self-invalidations, TSO-CC proposes to use *transitive reduction* of acquires. Every L1 maintains a monotonically increasing timestamp source, and these scalar timestamps are then associated with writes. This information can then be used to answer the question "is there potentially stale data in my cache?", and decide if self-invalidation of shared data is required. TSO-CC needs to apply transitive reduction at cache line granularity (writer timestamp per cache line), because TSO-CC cannot distinguish between synchronization and non-synchronization. If synchronization and non-synchronization data writes (that map to distinct cache lines) would share the same timestamp, *timestamp false sharing* would limit the effectiveness of the transitive reduction optimization.

Using the example in Fig. 1, TSO-CC ensures TSO as follows. ❶ $w \rightarrow w$ is ensured by with a shared directory as an arbitration point only permitting one writer at a time. Furthermore, increasing timestamps are assigned to the cache lines of data1, data2 and flag. ❷ Reading flag at event b_1 in processor B hits up to a maximum threshold, after which a miss is forced; this miss ensures that the most up-to-date value of flag is eventually read, and also causes self-invalidation of all other shared lines (in particular those containing data1 and data2). This miss also observes flag's timestamp, and processor B now associates this latest timestamp with processor A. Subsequent reads to data1 and data2 miss and obtain the (correct) up-to-date values, thereby ensuring $r \rightarrow r$; if self-invalidation had not taken place, processor B would have observed stale copies of data1 and data2, violating $r \rightarrow r$. Due to transitive reduction, the misses at b_2 and b_3 do not cause self-invalidation, as the timestamps of the received

data1 and data2 are both less than the timestamp of the already observed flag. Therefore, after the final event b_3, all of data1, data2 and flag are cached in B. To illustrate why maintaining timestamps at fine granularity is necessary, assume that processor A writes to several other locations following a_3, and these additional writes' timestamps are shared with data1 and data2, but B has not yet observed flag. In this case, the timestamp associated with the write of flag may be lower than that of data1 or data2, and these misses would in fact cause self-invalidation – this should be avoided by using timestamps at cache line granularity.

A major strength of TSO-CC's approach is that backwards compatibility is maintained with legacy TSO program codes. While TSO-CC is successful at limiting self-invalidations without explicit synchronization information, any additional information at the language level is still lost. Because of this, TSO-CC requires per cache line writer timestamps to achieve good performance. However, avoiding per cache line timestamps is highly desirable due to incurring storage overheads proportional to the number of lines. With RC3, we show that per cache line timestamps are unnecessary if synchronization operations are explicitly exposed.

III. x86-RCtso: Release Consistency for x86-64

With the extra data/synchronization information available at the language level, how to expose this information to existing architectures with stricter consistency models? We propose a solution for the widely deployed x86-64 architecture, via extending the memory model from TSO to *RCtso*, a memory model which differentiates data and synchronization. In extending the memory consistency model of an architecture, a major objective is to retain backwards compatibility with existing legacy codes as well as legacy microarchitectures (run new code on old systems). Specifically, in TSO [23] reads and writes already provide acquire and release semantics respectively. Therefore, reads and writes from legacy TSO codes must retain their original semantics.

Accordingly, we ensure that existing reads and writes retain acquire and release semantics, but add support for the missing relaxed *ordinary* memory reads and writes. The resulting memory consistency model is RCtso, which is similar to RCpc [16] in that it relaxes the $w_R \rightarrow r_A$ ordering, but unlike RCpc, requires *multi-copy atomicity* of read acquires and write releases, which it inherits from the original TSO implementation. Table I provides an overview of the ordering constraints enforced by RCtso.

Note that our variant of RCtso does not distinguish between special *sync* (acquire, release) and *nsync* operations [16]. Therefore, for ensuring correctness the compiler will have to treat nsync reads and nsync writes conservatively as sync acquires and sync releases respectively. This primarily concerns *racy programs*: these continue to work in RCtso, as long as racy accesses are marked as synchronization. In programmer-centric models, such as C11/C++11, such accesses require special annotation (e.g. C11/C++11 atomics) irrespective of the hardware level model, and correctness of such codes is not affected as long as the compiler provides a conservative mapping to acquires/releases.

A. ISA extension details

This section describes the details of an ISA extension for the x86-64 architecture, effectively changing the supported memory consistency model from x86-TSO to x86-RCtso. In order to add the proposed relaxed ordinary memory operations, we have to label them explicitly. We can do so using instruction prefixes for memory operations. In x86-64 a group of prefixes, which were previously used for 32-bit mode to denote segment register overrides (CS, DS, ES, SS), have become unused and their meaning was changed to null prefixes [31] (§B.7).

Any one of these prefixes can be reused and their semantics changed from null to denote relaxed *ordinary* memory operations. In doing so, the ISA would not break compatibility with existing legacy codes, as unprefixed loads and stores retain their acquire and release semantics; these programs would merely impose a stricter program ordering among instructions. This also means that existing legacy synchronization libraries are compatible with new codes that make use of the extension to RCtso. Finally, this approach also ensures that new codes targeting RCtso are compatible with legacy microarchitectures, as in this case the prefix would revert to a null prefix. Therefore, the imposed program ordering will only be stricter than required, ensuring correctness [5].

IV. RC3: Protocol Design

Our primary technical contribution is the RC3 protocol which takes advantage of the explicit labelling. This section describes the detailed protocol design: first we give an overview of the protocol (§IV-A); this is followed by a detailed description of the basic RC3 protocol without optimizations (§IV-B), and continue extending the basic protocol with the transitive reduction (§IV-C) and shared read-only optimizations (§IV-E). Throughout, the organization chosen assumes private L1 caches per core, and a tiled (NUCA) shared L2 with an embedded directory (§IV-G).

A. Overview

We base the protocol on TSO-CC, as outlined in §II-B, and modify the protocol to exploit the fact that RCtso conveys synchronization and ordinary operations to the hardware explicitly. By exploiting this additional information, our goal is to further reduce the storage overheads of the resulting RC3 protocol, but retain comparable performance characteristics.

As the protocol is already aware of acquires and releases, we add support for the new ordinary memory operations. Upon acquires, where the last writer is not the requester, the protocol self-invalidates all shared cache lines in the local

Table I
RCTSO ORDERING REQUIREMENTS

happens-before $\nrightarrow$	Read-Acquire (r_A)	Write-Release (w_R)	Read-Ordinary (r)	Write-Ordinary (w)
Read-Acquire (r_A)	X	X	X	X
Write-Release (w_R)		X		
Read-Ordinary (r)		X		
Write-Ordinary (w)		X		

cache. It is worth noting, however, that self-invalidation is not required upon ordinary reads. Furthermore, shared lines fetched by ordinary memory operations can hit indefinitely in the local caches. In addition, we retain the TSO-CC optimization which permits acquires to hit shared cache lines up to a maximum number of accesses, as this ensures adequate performance for legacy codes.

In order to achieve good performance, TSO-CC proposes the transitive reduction optimization at cache line granularity. This is necessary, as newer writes (to different cache lines) after a write release will be assigned increasing timestamps, but each write retaining a distinct timestamp value (until another write to the same line) due to using timestamps at cache line granularity. As timestamps assigned to older write releases on different cache lines are unaffected until another release, unnecessary self-invalidations are rare. TSO-CC is effectively sharing timestamps at cache line granularity; at this granularity *timestamp false sharing* can only happen for all addresses mapped to a single cache line.

With RCtso, however, the protocol is explicitly conveyed information about synchronization and data accesses, and because data accesses dominate, self-invalidation is suppressed for these accesses regardless. In the earlier example (§II-B) illustrated with Fig. 1, using RCtso allows the read in B of flag to be marked as an acquire, and data1/data2 marked as ordinary reads. In this case, self-invalidation can only take place at b_1, but not b_2 or b_3 even if the timestamps of data1/data2 were higher than of flag – assuming shared timestamps and continued writes after the release of flag in A. Therefore, maintaining timestamps at cache line granularity is overkill, as explicit synchronization is infrequent and limited to relatively few addresses for which timestamps can be shared. Our hypothesis is, that applying timestamps at entire address-space granularity with RCtso optimized workloads is sufficient to realize the same performance benefits of transitive reduction as TSO-CC (validated in §VI-B). Consequently, we can eliminate per cache line timestamps from L1s and L2 tiles, and only require maintaining per-L1 timestamps. In particular, per-L1 timestamps are still effective at reducing redundant acquires, e.g. due to conservative synchronization and acquiring mostly shared read-only data.

Furthermore, the protocol requires changes to the shared read-only optimization, as per cache line timestamps were previously used to *decay* lines from shared-written back to shared read-only. Our approach here is to reuse data structures already present in TSO-CC, but used for timestamp resets; specifically, we reuse the epoch-id, and only maintain epoch-ids per L2 cache lines to identify that a period of time has elapsed since the last write. As the epoch-ids require substantially less space then timestamps, this optimization, given its performance benefits, can be justified.

B. Basic protocol

The following outlines the stable states, actions and transitions of the protocol.

Stable states: The protocol distinguishes between invalid, private and shared states. Cache lines in the L1 can be in invalid (Invalid), private (Exclusive, Modified, Exclusive_L, Modified_L) and shared (Shared, Shared_L) states. In the L2, private (Exclusive) cache lines only require a pointer b.owner to the current owner; shared (Shared) cache lines are untracked in the L2, and do not require tracking a list of sharers. The L2 maintains an additional state Uncached for cache lines not present in any L1, but valid in the L2.

We must introduce pairs of states in the L1: the base state, and a state (∗_L) denoting the line was fetched due to a reLaxed ordinary memory operation. This distinction is required to deal with cases where an ordinary memory operation caused a miss, but followed by a synchronization hit. In the following we refer to the set of states with a common label prefix as Prefix∗, e.g. the set of states Exclusive and Exclusive_L are referred to as Exclusive∗. A transition from Exclusive∗ to Modified∗ means the transition is to the state with the same suffix (if any).

Read-Ordinary: Read requests (GetS) to cache lines invalid in the L2 cause an Exclusive response to the requesting L1, which must then acknowledge the response and transitions to Exclusive_L. If the cache line is in state Exclusive in the L2, the GetS request is forwarded to the current owner. The owner will then downgrade its copy from Exclusive∗ or Modified∗ to Shared∗. The owner responds to the initial requester with the data, which transitions to Shared_L; the owner additionally sends acknowledgement (if Exclusive∗) or data (if Modified∗) to the L2, which transitions the cache line to the Shared state. On subsequent read requests to the L2, the L2 responds with Shared data. Ordinary read accesses to Exclusive∗, Modified∗, and Shared∗ cache lines *always hit* in the L1.

Read-Acquire: Similarly to an ordinary read operation, a GetS request is sent to the L2. Upon receipt of a response, the L1 transitions to the respective base state, Exclusive or Shared.

As shared lines are untracked in the L2, all shared lines in the L1 must eventually be self-invalidated. To maintain the

$r_A \rightarrow r$ and $r_A \rightarrow r_A$ ordering, L1s *self-invalidate all Shared∗ cache lines after every L1 synchronization miss, where the transition is to a base state, and the response's last writer is not the requesting L1*.

Read acquire accesses hit to private lines (Exclusive_L, Modified_L) fetched due to an ordinary memory accesses, but are forced to perform self-invalidation of shared cache lines, as ordinary reads do not cause self-invalidation. This is, as outlined above, to address the corner case where an ordinary memory operation fetched a cache line, but the same cache line is subsequently accessed by a synchronization operation. After self-invalidation, the cache line is transitioned to the base state (e.g. from Exclusive_L to Exclusive). A read acquire accessing a cache line in Shared_L causes a miss, as the cache line is most likely stale.

Read acquire accesses to Shared cache lines are allowed to hit, but only up to a predefined maximum number of accesses, at which point a miss is forced. This requires extra storage for the access counter b.acnt. We reuse this optimization from TSO-CC, as firstly it provides adequate performance for legacy codes optimized for TSO. Secondly, this is the reason why the $w_R \rightarrow r_A$ ordering is relaxed in RC3, and thus targets RCtso.

Write-Ordinary: An ordinary write operation can only hit in the L1 if the line is held in the Exclusive∗ or Modified∗ states. Transitions from Exclusive∗ to Modified∗ are silent. An ordinary write misses in the L1 in any other state, causing a GetX request sent to the L2. Upon receipt of a response, the local cache line's state changes to Modified_L, the data is written to the L1, and an acknowledgement is sent to the L2. The L2 cache updates the cache line's state to Exclusive and updates b.owner with the requester's id.

If another L1 requests write access to a private line, the L2 forwards the request to the owner stored in b.owner, which then invalidates the line and passes ownership to the requester. Since the L2 only responds to write requests if it is in a stable state, i.e. it has received the acknowledgement of the last writer, there can only be one writer at a time. This serializes all writes to the same address at the L2 cache.

Upon a write request to a Shared line, the L2 immediately responds with a data response message and transitions the line to Exclusive. Note that even if the cache line is in Shared, the L2 must send the entire line, as the requesting core may have a stale copy. On receiving the data message, the L1 transitions to Modified_L either from Invalid or Shared∗. Note that there may still be other copies of the line in Shared∗ states in other L1 caches, but since they will eventually miss due to self-invalidation, and also cause self-invalidation of shared lines on synchronization misses, RCtso is satisfied.

Write-Release: Write releases hit in the same states as ordinary writes. Given $w_R \rightarrow r_A$ is relaxed, hits in the Exclusive_L or Modified_L states do not cause self-invalidation, and are treated as in the ordinary write case. Upon a write release miss, the final state upon receipt of a response is Modified; as per the rules outline above, such a miss would also cause self-invalidation.

Evictions: Inclusivity must be maintained for cache lines which are tracked by the L2: on evictions from the L2, evictions from Exclusive (and later SharedRO, see §IV-E) require invalidation requests to the owner. Shared lines are untracked, and therefore evicted silently from the L2. Evictions from the L1 in states Exclusive∗ and Modified∗ require updating the L2 accordingly, which then transitions the line to Uncached; Shared∗ lines are evicted silently.

C. Opt. 1: reducing self-invalidations of redundant acquires

In order to satisfy the $r_A \rightarrow r$ ordering, the basic protocol applies self-invalidation of Shared∗ lines at L1 misses. However, subsequent acquires would always cause self-invalidation. If a release has already been observed, and all memory operations before it have previously been made visible via self-invalidation, self-invalidating again – upon acquiring the same, or any release that happened before it – is not required. To reduce unnecessary invalidations, we apply a variant of transitive reduction [32] like TSO-CC, but limited to synchronization misses.

Each L1 maintains a local *current timestamp* cur_ts of fixed size. The size of the timestamp depends on the storage requirements, but also affects the frequency of the timestamp resets, which is discussed in more detail in §IV-D. The L1 local timestamp must be *incremented on every release*.

Upon propagating a cache line to the L2 cache, the L1's current timestamp cur_ts is propagated. The L2 then updates its respective entry for the sender in a last-seen timestamp table ts_L1. Note that, if we have multiple L2 tiles, the protocol requires a timestamp table per L2 tile. Each L1 also maintains a last-seen timestamp table ts_L1. The maximum possible entries per timestamp table can be less than the total number of cores, but will require an eviction policy to deal with limited capacity. The L2 responds to requests with the data, the writer b.owner and the last writer's most recent timestamp ts_L1[b.owner].

Thus, to reduce invalidations, only where the *L2's last-seen timestamp is larger than the L1's last-seen timestamp of the writer of the requested line*, treat the event as a *true acquire* and self-invalidate all Shared∗ lines.

For those data responses where the timestamp is invalid (never written to since the L2 obtained a copy) or there does not exist an entry in the L1's timestamp-table (never read from the writer before), a self-invalidation is necessary. This is because timestamps are not propagated to main-memory and it may be possible for the line to have been modified and then evicted from the L2.

In case of an ordinary access miss followed by a read acquire hit to the same line, transitive reduction cannot be directly applied (since the second access being a hit does not involve a response with a timestamp). However, we can still apply the transitive reduction as follows: on an ordinary

access response, we check for *true acquire*, and if the check *would have caused self-invalidation*, we proceed to transition to the relaxed state, otherwise to the corresponding base state. This may still cause unnecessary self-invalidations where a synchronization miss (timestamp larger than last seen, causes self-invalidation) to a different line happens between the ordinary miss and the acquire hit (timestamp would have been less than or equal to last seen). Fortunately, this case is infrequent according to our evaluation.

D. Timestamp resets

Because timestamps are finite, we have to deal with timestamp resets. Given the maximum timestamp size is chosen appropriately, and as they are only incremented on releases, resets should occur infrequently. If the current timestamp cur_ts is exhausted, L1s must broadcast a timestamp reset message to all L1s and L2 tiles. Upon receiving a timestamp reset message, a L1 invalidates the sender's entry in the timestamp table ts_L1; similarly for each L2 tile.

Handling races: It is possible for timestamp reset messages to race with data request and response messages: the case where a data response with a timestamp from a previous epoch arrives at a L1 which already received a timestamp reset message needs to be accounted for. The protocol requires maintaining *epoch-ids* per L1. The epoch-id of a L1 is incremented on every timestamp reset and the new epoch-id is sent along with the timestamp reset message. It is not a problem if the epoch-id overflows, as the only requirement for the epoch-id is to be distinct from its previous value. However, we assume a bound on the time it takes for a message to be propagated, and it is not possible for the epoch-id to overflow and reach the same epoch-id value of a message in transit.

Each L1 and L2 tile maintains a table of epoch-ids for every other L1. Every data message that contains a timestamp, must now also contain the epoch-id of the source of the timestamp. Upon receipt of a data message, the L1 compares the expected epoch-id with the data message's epoch-id: if they do not match, the same action as on a timestamp reset has to be performed, and can proceed as usual if they match.

Epoch optimization: As the current epoch is known to L2 tiles via the epoch-id table they maintain, we can make use of the epoch-id information to convey a more precise ordering than simply responding with the last-seen timestamp. This optimization requires addition of a small amount of extra storage for the written epoch-id to each L2 cache line.

If we know that the last writer's current epoch-id is different from the L2 cache line's epoch-id, the write must have happened before the last timestamp reset. In this case, the cache line's window for assigning the last-seen timestamp has *expired*. Upon cache line expiry, it is sufficient to assign the smallest valid timestamp to the response, so that we can avoid self-invalidation where the release has happened before the most recent release – under the assumption that the requesting L1 has already seen a more recent timestamp from the last writer.

One additional case must be dealt with: if the smallest valid timestamp is used in case of cache line expiry, it should not possible for a L1 to skip self-invalidation due to the line's timestamp being equal to the smallest valid timestamp. To address this case, the next timestamp assigned to a request response after a reset must always be larger than the smallest valid timestamp.

E. Opt. 2: shared read-only with epoch based decay

The basic protocol suffers from a pathological case, where shared cache lines which are written to very infrequently but read frequently are self-invalidated unnecessarily. TSO-CC greatly benefits from introducing the shared read-only optimization to avoid this, but makes use of per cache line timestamps in deciding when a shared line should be classified read-only. This section describes an alternative policy without full timestamps.

We add another state SharedRO for shared read-only cache lines, which are excluded from self-invalidation. A line transitions to SharedRO instead of Shared if the line is not modified by the previous Exclusive owner. Additionally, cache lines in the Shared state are transitioned (*decay*) to SharedRO upon *expiry*: *if the cache line's written epoch does not equal the last writer's current epoch* (see §IV-D). The L1s maintain SharedRO and SharedRO_L states, where the request was either due to synchronization or an ordinary operation respectively. On an acquire to a SharedRO_L line, the L1 must self-invalidate shared lines, followed by the line transitioning to SharedRO – as described above, the prior ordinary access does not cause self-invalidation.

In the case of a synchronization access to a SharedRO cache line where the last writer is not known, the L1 would always have to perform self-invalidation. Similar to TSO-CC, we can introduce L2 SharedRO timestamps, where each L2 maintains a current timestamp. As we do not store timestamps in cache lines, a SharedRO response is assigned the L2's current timestamp. On a cache line transitioning from Exclusive or Shared to SharedRO, the L2 tile increments its current timestamp. Each L1 must maintain a table ts_L2 of last-seen timestamps for each L2 tile. On receiving a SharedRO response from the L2, the following rule determines if self-invalidation must occur: if the line's timestamp is *larger than the last-seen timestamp from the L2*, self-invalidate all Shared$*$ lines. Furthermore, to reduce the number of L2 timestamp increments, the L2's current timestamp is not incremented if there does not exist a cache line which transitioned (since the last increment) to a state from which SharedRO can be reached (for the specific rules, see [24]).

Upon resetting a L2 tile's timestamp, a broadcast is sent to every L1, and the L1s remove the entry in ts_L2 for the sending tile. As outlined in §IV-D, to avoid races, L2s also

Table II
RC3 SPECIFIC STORAGE REQUIREMENTS

L1	**Per node:** • Current timestamp cur_ts, B_{ts} bits • Current epoch-id cur_eid, $B_{epoch-id}$ bits • Timestamp-table ts_L1$[n]$, $n \leq C_{L1}$ entries • Epoch-ids epoch_ids_L1$[n]$, $n = C_{L1}$ entries Only required if SharedRO opt. (§IV-E) is used: • Timestamp-table ts_L2$[n]$, $n \leq C_{L2-tiles}$ entries • Epoch-ids epoch_ids_L2$[n]$, $n = C_{L2-tiles}$ entries **Per line b:** • Number of accesses b.acnt, B_{maxacc} bits
L2	**Per tile:** • Last-seen timestamp-table ts_L1$[n]$, $n = C_{L1}$ entries • Epoch-ids epoch_ids_L1$[n]$, $n = C_{L1}$ entries Only required if SharedRO opt. (§IV-E) is used: • Current timestamp, B_{ts} bits • Current epoch-id, $B_{epoch-id}$ bits • Increment-timestamp-flags, 2 bits **Per line b:** • Epoch-id b.epoch_id, $B_{epoch-id}$ bits • Owner (Exclusive), last writer (Shared), coarse vector (SharedRO) as b.owner, $\lceil log(C_{L1}) \rceil$ bits

maintain epoch-ids, and every L1 maintains a table of epoch-ids epoch_ids_L2. To avoid sending larger timestamps than the current timestamp, we again apply the epoch optimization.

Writes to SharedRO: A write request to a SharedRO line triggers broadcast invalidate, and subsequent acknowledgements. Network traffic can be reduced by reusing the b.owner bits as a broadcast filter [24]. SharedRO evictions from L1 are therefore silent, but evictions from L2 requires broadcast invalidate, followed by acknowledgements.

F. Atomic instructions & fences

Implementing atomic read and write instructions, such as RMWs, is trivial with the proposed protocol: each atomic instruction issues a GetX request. Fences require unconditional self-invalidation of cache lines in the Shared state. Note that in our implementation, fences do not invalidate cache lines fetched by ordinary memory operations (Shared_L), which implies that fences do not enforce ordering between ordinary memory operations.

G. Storage requirements & organization

Table II shows a detailed breakdown of storage requirements for RC3, referring to literals that have introduced throughout §IV. We assume a local L1 cache per core and a NUCA [33] architecture for the shared L2 cache.

While we chose a simple sparse directory embedded in the L2 cache for all configurations (§VI-A) used in the evaluation, our protocol is independent of a particular directory organization. It is possible to further optimize our overall scheme by using directory organization approaches such as in [11], [14]; however, this is beyond the scope of this paper. Also note that the protocol does not require inclusivity for Shared$*$ lines, alleviating some of the set conflict issues associated with the chosen organization.

By eliminating per cache line timestamps, we significantly simplify cache organization compared to TSO-CC. Notably, eliminating per cache line timestamps simplifies lookup of the timestamps as they no longer need to be associated with a particular address tag. Other structures such as the MSHR also no longer require a timestamp entry.

Table III
SYSTEM PARAMETERS

Core-count & frequency	32 (out-of-order) @ 2GHz
Write buffer entries ROB entries	32, FIFO 40
L1 I+D -cache (private) L1 hit latency	32KB+32KB, 64B lines, 4-way 3 cycles
L2 cache (NUCA, shared) L2 hit latency	1MB$\times$32 tiles, 64B lines, 16-way 30 to 80 cycles
Memory Memory hit latency	2GB 120 to 230 cycles
On-chip network	2D Mesh, 4 rows, 16B flits
Kernel	Linux 2.6.32.61

Table IV
BENCHMARKS AND THEIR INPUT PARAMETERS

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

V. EVALUATION METHODOLOGY

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

A. Simulation Environment

We use the Gem5 simulator [34] with Ruby and GARNET [35] in *full-system* mode. The ISA is x86-64 with RCtso extensions added (§III). The processor model used for each core is a simple out-of-order processor. Table III shows the key-parameters of the system. As the protocols evaluated explicitly allow accesses to stale data, we added support to the simulator to functionally reflect cache hits to stale data; unmodified, the used version of Gem5 in full-system mode would assume the caches to always be coherent otherwise.

Table V
COHERENCE STATE STORAGE OVERHEADS WITH ALL OPTIMIZATIONS ENABLED: PRIVATE L1 PER CORE, 1MB PER L2 TILE, AND AS MANY TILES AS CORES; THE TIMESTAMP-TABLE SIZES MATCH THE NUMBER OF L1S AND L2 TILES; $B_{epoch-id} = 3$ BITS PER EPOCH-ID. NORMALIZED W.R.T. MESI, COHERENCE STORAGE MB.

Cores	32	64	128
MESI	100% (2.13)	100% (8.27)	100% (32.53)
TSO-CC	62% (1.33)	34% (2.80)	18% (5.91)
RC3	34% (0.73)	19% (1.59)	11% (3.49)
RC-base	24% (0.52)	14% (1.16)	8% (2.56)

B. Workloads

Table IV shows the benchmarks we have selected from the PARSEC [36], SPLASH-2 [37] and STAMP [38] benchmark suites. The STAMP benchmark suite has been chosen to evaluate transactional synchronization compared to the more traditional approach from PARSEC and SPLASH-2; the STM algorithm used is NOrec [39] as it is the current default.

In order to optimize the full-system software stack we use, we modified GCC's machine description for x86-64, which adds the chosen prefix (SS prefix) for all ordinary data operations. As all chosen workloads make clear use of synchronization libraries, we only had to make sure the synchronization libraries were unmodified, in effect using read acquires and write releases. We further optimized as many system libraries of the distribution as possible.

All selected workloads correctly run to completion with the evaluated protocol configurations. The program codes are unmodified, but targeting x86-64 with RC extensions (§III). The Linux kernel used, however, is unmodified and compiled *without* RC extensions, as we ran into limitations of our ad-hoc conversion from TSO to RCtso. This means that our results are conservative, and a system with a fully optimized software stack will yield the same or better performance as our evaluation shows. A rigorous conversion of x86-TSO optimized codes to x86-RCtso is beyond the scope of this paper. With our conversion, the total size of all workload binaries increases by 7%.

VI. EXPERIMENTAL RESULTS

The goal of our evaluation is to analyze the storage (§VI-A) and performance characteristics (§VI-B) of RC3 in comparison with MESI, a conventional RC baseline and TSO-CC.

A. Protocol configurations & storage

We have chosen the MESI directory protocol implementation part of Gem5 as the baseline; this implementation provides a fair baseline as it is used by several related works, also as part of the original Wisconsin GEMS simulation toolset [40]. With regard to TSO-CC, we have chosen the same parameters as the ones determined optimal in the limited design space exploration of [24].

We include the shared read-only optimization as described in §IV-E in all configurations (except MESI, which is unmodified). Note that, we do not include a version of RC3 with infinite timestamps, as this renders the SharedRO *decay* optimization ineffective due to non-resetting timestamps (epoch-id never changes). Our evaluation showed that a version of RC3 with infinite timestamps performs worse than or equal to a configuration of RC3 with finite timestamps – as such, we exclude this configuration. Below we consider the following configurations: RC-base, TSO-CC, RC3.

RC-base: A conventional RC protocol that removes the sharer list, and relies on self-invalidation of shared cache lines on acquires. Ordinary read misses do not cause self-invalidation. We derive RC-base's implementation from RC3, effectively a version without timestamps. This is to provide a fairer comparison, in particular so that RC-base's implementation includes the shared read-only optimization (however, lacking timestamps, without the ability to decay Shared lines). In this protocol, acquires always miss if the cache line is in Shared state. With the evaluated system configuration as seen in Table III, RC-base reduces coherence storage requirements by 76% compared to MESI for 32 cores.

TSO-CC: This version is the overall best performing TSO-CC configuration as found in [24]. This configuration uses 4 bits for the accesses counter, 12 bits for the timestamps and a 3 bit write-group counter. TSO-CC reduces storage requirements by 38% compared to MESI for 32 cores.

RC3: This is the RC3 protocol with all optimizations enabled. This configuration uses 4 bits for the accesses counter and 12 bit timestamps. Compared to MESI for 32 cores, this configuration of the RC3 protocol saves 66% on-chip storage, and 45% compared to TSO-CC. In addition to purely saving storage overheads, RC3 simplifies cache organization compared to TSO-CC, thereby saving power consumption; however, a detailed study of power consumption is beyond the scope of this paper and reserved for future work. We include **RC3-legacy** to show the performance of *legacy codes* with the RC3 protocol. In this configuration, the ISA extension is not used for the workloads.

Table V shows a comparison of the extra coherence storage requirements between MESI, TSO-CC, RC3 and RC-base (in order of decreasing storage requirements). With the chosen configurations, RC3 reduces on-chip storage requirements by 89% (41%) over MESI (TSO-CC) for 128 cores.

B. Performance Results

Our initial hypotheses are as follows. Firstly, we expect that RC3, with the help of the transitive reduction optimization (albeit with per-L1 timestamps), will perform significantly better than RC-base. Secondly, despite using only per-L1 timestamps, we expect RC3 to perform as well as TSO-CC (which uses per cache line timestamps), as it can additionally leverage explicit synchronization information. In the following, we will validate our hypotheses by comparing

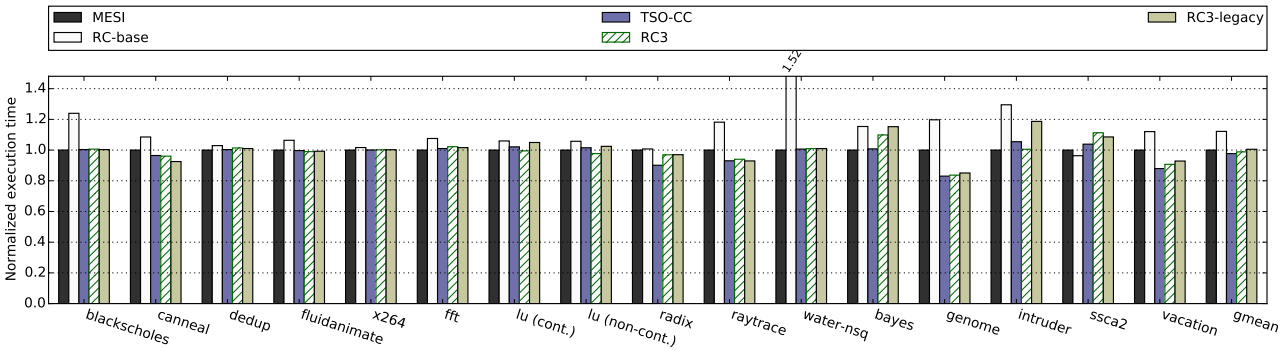

Figure 2. Execution times, normalized against MESI.

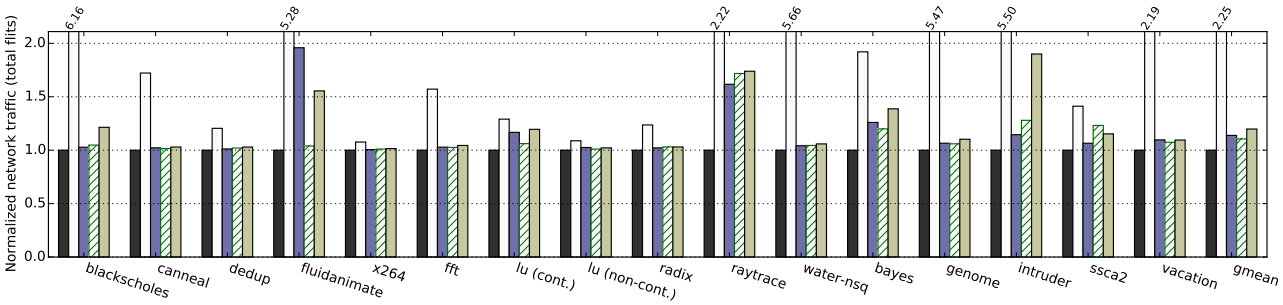

Figure 3. Network traffic (total flits), normalized against MESI.

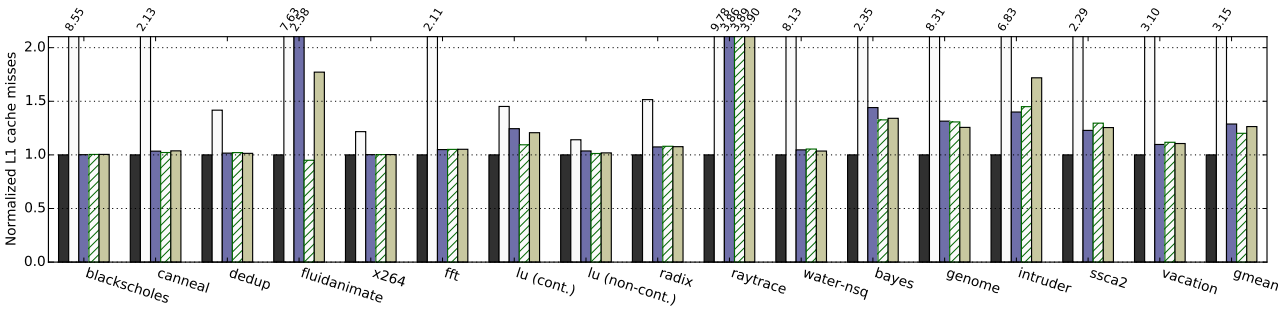

Figure 4. L1 cache misses, normalized against MESI.

the performance and network overhead of RC3 with that of RC-base and TSO-CC (and MESI). In order to isolate the contribution of synchronization information in RC3, we will also compare against RC3-legacy, which is identical to RC3 in all respects, except that it is not conveyed explicit synchronization information. The analysis focuses on performance results in Fig. 2 showing execution times, and Fig. 3 showing network traffic; we use supporting data from Figures 4 and 5 which show cache hit/miss rates, and Fig. 6 showing total self-invalidations.

With explicit synchronization information, is transitive reduction using timestamps still required for performance? In order to answer this question, we compare the performance of RC-base with that of RC3. As seen in Fig. 2, on average the baseline RC protocol RC-base causes a slowdown of 12% compared to MESI. Network traffic (Fig. 3) is far more sensitive, with an average increase of 125% compared to MESI. Interestingly, the network traffic as well as L1 misses (Fig. 4) are heavily correlated, yet often with much less noticeable effects on execution times, as the out-of-order cores can hide miss latencies well. Introducing the optimizations of RC3 provides an average improvement over RC-base of 12% in terms of execution times, and 57% in terms of network traffic. RC3 reduces redundant acquires via the transitive reduction optimization, and most of the difference can be attributed to the consequent reduction of self-invalidations: compared to RC-base we note a reduction of self-invalidations by 800% on average. However, why does RC3 perform poorly in the first place with respect to self-invalidations? We believe this is due to redundant acquires in RC-base, an artifact of overly conservative synchronization in parallel codes [41]. RC3 solves this problem via transitive reduction. This validates our first hypothesis, that RC3 outperforms RC-base, and therefore transitive reduction improves performance even where explicit synchronization

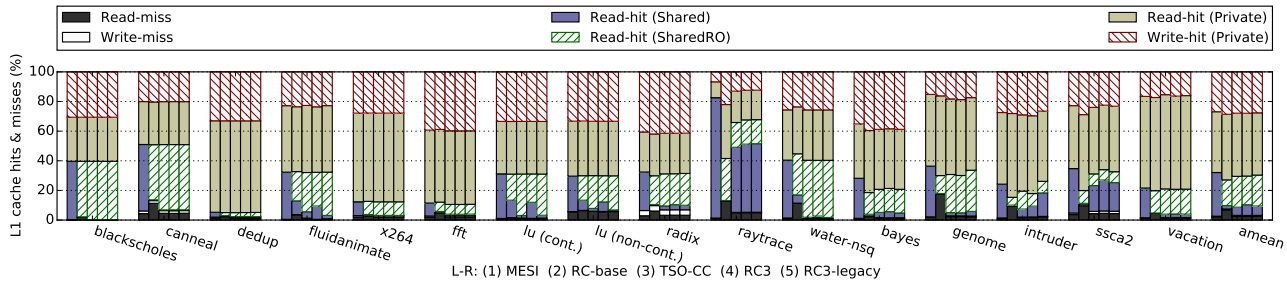

Figure 5. L1 cache hits and misses; hits split up by Shared, SharedRO and private (Exclusive, Modified) states.

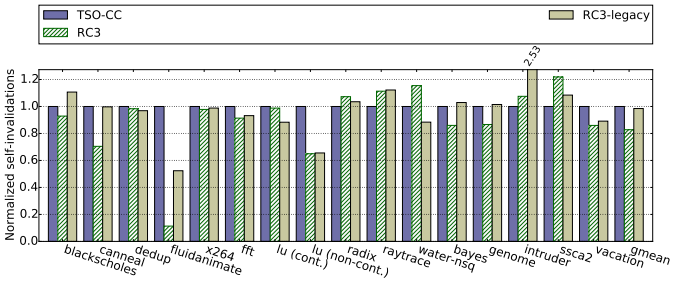

Figure 6. Normalized self-invalidations w.r.t. TSO-CC.

information is provided to the protocol.

We note that write misses do not vary much across configurations, and the biggest difference in performance is due to read misses. Firstly, this is due to the fraction of reads (avg. 70%) dominating that of writes (avg. 30%), and secondly because writes are not in the critical path as they are entered into a write buffer. Furthermore, write misses due to downgrades are infrequent because of relatively small number of communicating accesses, in particular for SPLASH-2 and PARSEC benchmarks [42].

With explicit synchronization information, how does removing per cache line timestamps (and instead only rely on per-L1 timestamps) affect performance? In order to answer this question, we compare the performance of RC3 with that of TSO-CC (and MESI). RC3 performs as well, in terms of execution times, as TSO-CC and the MESI baseline on average. The best case execution time is achieved with *genome*, which improves by 16% over the MESI baseline; in the worst case we observe a slowdown of up to 11% for *ssca2*. Fig. 6 shows total self-invalidations normalized against TSO-CC: on average, RC3 self-invalidates 17% fewer cache lines compared to TSO-CC. By reducing self-invalidations, RC3 reduces L1 misses (Fig. 4) by 7% compared to TSO-CC; thereby RC3 reduces network traffic by 3% over TSO-CC. This validates our second hypothesis that RC3 performs at least as well as TSO-CC and MESI; even though RC3 only uses per-L1 timestamps, it leverages explicit synchronization information to self-invalidate fewer cache-lines.

Without explicit synchronization information, how does removing per cache line timestamps (and instead only rely on per-L1 timestamps) affect performance? To answer this, we compare the performance of RC3-legacy with that of TSO-CC and RC3. On average, execution times of RC3-legacy are very close to TSO-CC and RC3, but network traffic increased by 5% and 8% respectively. Indeed, self-invalidations (Fig. 6) appear to be on-par with TSO-CC, but 20% higher than RC3. However, we see higher variance across benchmarks. In particular for some STAMP benchmarks, RC3 is significantly better – in the case of *intruder*, RC3-legacy increases execution time by 17% and network traffic by 48%.

From this study we can observe that, for workloads with relatively frequent synchronization such as *intruder* and *bayes* in STAMP, more precisely identifying synchronization either via exposing synchronization (RC3) or using fine grained timestamps (TSO-CC) is important. However, for other benchmarks (e.g. most from PARSEC and SPLASH-2), where time spent communicating is relatively low, even with per-L1 timestamps but no explicit synchronization information (RC3-legacy), performance is good. In these cases, the protocol is efficient at properly classifying (see Fig. 5) private and shared read-only data which are excluded from self-invalidation.

VII. Related Work

RCtso has not been explicitly mentioned in the literature, although variants of the RC memory model have been formally defined in the literature [27]. In particular, the RCpc consistency model, also relaxes the release to acquire ordering; in contrast to RCtso, however, RCpc but does not require multi-copy atomicity among releases and acquires. While not explicitly referred to as RCtso, Intel Itanium implements what we consider RCtso [43].

The definitions of Adve et al.'s *data-race-free* [25], [26], [29] and Gharachorloo et al.'s *properly labelled* [16], [27] form the basis for the programmer-centric approach we use in our discussion to highlight the fact that the programmer does not need to be exposed to the complexity of the resulting hardware level consistency model. Our work takes a more practical approach, proposing a detailed implementation of the memory consistency model in an existing architecture, and how the previously stricter (x86-TSO) consistency model can be extended (x86-RCtso).

Some existing architectures have started to provide support for achieving a mapping from a language level model to the hardware memory model, that lets it retain synchronization

information. For example, the ARMv8 architecture [44] has introduced releases and acquires into the ISA. In contrast with ARM, where the resulting extended model (via adding releases and acquires) is stronger than the original model, the case for x86 is more challenging as the extended model RCtso is weaker than the original model; legacy issues arise in the latter but not the former.

Note also that recent Intel processors have introduced hardware transaction extensions (including XACQUIRE, XRELEASE) [45]. However, these are for a different purpose, namely lock elision [41]. Our problem is orthogonal, as we are interested in weakening the memory consistency model; in this instance we argue that since TSO reads and writes already have acquire and release semantics respectively, exposing relaxed memory operations is the right approach. It is worth noting that in the same way as hardware transaction extensions were introduced in a backwards compatible way, we propose reuse of a null prefix on memory operations to introduce more relaxed ordinary memory operations.

Consistency directed coherence: Several recent works [17], [18], [19], [20], [21], [22] target relaxed memory consistency models, typically RC or Weak Ordering [5]; DeNovo [18] and DeNovoND [22] follow a programmer-centric approach (SC for DRF). These works introduce a number of optimizations for enhancing the performance of relaxed consistency protocols. Notably, optimizing higher-level synchronization primitives (locks, barriers, etc.) [18], [22], [46] would help improve latencies and reduce misses, as polling behaviour could be avoided. These optimizations, however, are orthogonal to our proposal, as we stuck with implementations of current operating system and standard library vendors. Unfortunately, none of these approaches can directly be applied to existing architectures with stricter models.

SPEL [47] is a dual-consistency protocol, which can guarantee SC, and provide performance improvements given explicit code annotations denoting DRF. Although legacy compatible, the protocol does not reduce storage overheads.

Coherence for GPUs has become a recent topic of interest, to more efficiently support wider ranges of workloads. GPUs are typically programmed using higher level languages (e.g. OpenCL), and the vendor is responsible for a correct mapping to the hardware level. As such, the system-centric memory consistency model of GPUs has not been readily exposed. However, recent proposals for coherent memory systems on GPUs propose RC [48], [49].

As referred to in previous sections, TSO-CC [24] is the most closely related protocol; however, we eliminate the per cache line timestamp requirement by relying on RCtso explicitly distinguishing synchronization and data operations.

Data structures in eager protocols: Numerous works attack the cache coherence problem by optimizing the data structures and cache & directory organization to maintain coherence state – in particular the list of sharers more efficiently via: hierarchical directory organizations [13], [15]; sharing vector compression [12], [50]; variable size sharing vectors [14]; or optimizing directory utilization [11], [14].

While most of these approaches are not directly applicable in protocols without a list of sharers, some can also be applied to different protocols (such as the proposed RC3) – in particular those that optimize directory utilization (e.g. Cuckoo [11]). None of these approaches consider the memory consistency model explicitly. Unlike these approaches, we propose changing the protocol, and by optimizing for the memory consistency model, to only require less costly data structures in the first place. By combining directory optimization approaches and the RC3 protocol, the potential on-chip storage savings can be even greater.

VIII. Conclusion

In recent years we have seen widespread convergence towards clearly defined programming language level memory consistency models. Each of these models requires the programmer to explicitly distinguish between synchronization and data memory operations. If such a language level model is mapped to a compatible lower level hardware consistency model that preserves the synchronization information, the additional information can then be exploited by hardware for enhanced performance and scalability. In particular, we have seen a resurgence on the study of lazy cache coherence protocols that exploit this explicit labelling of synchronization and data to achieve scalable coherence protocols. Most of these proposals assume (variants of) RC, which inherently differentiates between data and synchronizations.

There are however existing architectures, which support hardware consistency models that do not directly allow for synchronization information to be conveyed. To make matters worse, some of these architectures (most notably x86) support stricter models. It is not possible for such architectures to transition to RC overnight, as legacy code written assuming the stricter model should continue to work. This paper has presented a viable way to achieve this transition for x86-64, by addressing: ① how synchronization information from the language level can be exposed to the hardware; and ② how cache coherence can take advantage of this information.

We have shown synchronization information can be conveyed relatively easily (and elegantly) by simply conveying whether or not a memory operation is a relaxed operation using unused prefixes in the ISA. We have then shown how the cache coherence protocol can be designed to take advantage of the relaxations, yet ensure TSO for legacy codes. All this with significant storage savings in comparison to not only MESI but also TSO-CC, a lazy coherence protocol designed to target TSO. Performance of RC3 is significantly better than baseline RC as we eliminate redundant acquires. Despite using only per-L1 timestamps (as opposed to per cache line timestamps employed by TSO-CC), RC3's performance is comparable to TSO-CC (and MESI) as we exploit synchronization information.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers (and our shepherd Mattan Erez) for their helpful comments and advice. This work is partly funded by EPSRC grants EP/M027317/1 and EP/M001202/1 to the University of Edinburgh.

REFERENCES

[1] *Programming Languages — C*, 2011, ISO/IEC 9899:2011. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1539.pdf.

[2] *Programming Languages — C++*, 2011, ISO/IEC 14882:2011. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[3] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *PLDI*, 2008, pp. 68–78.

[4] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," in *POPL*, 2005, pp. 378–391.

[5] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[6] M. Dubois, C. Scheurich, and F. A. Briggs, "Memory Access Buffering in Multiprocessors," in *ISCA*, 1986, pp. 434–442.

[7] L. I. Kontothanassis, M. L. Scott, and R. Bianchini, "Lazy Release Consistency for Hardware-Coherent Multiprocessors," in *SC*, 1995, p. 61.

[8] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," in *ISCA*, 1995, pp. 48–59.

[9] C. Scheurich and M. Dubois, "Correct Memory Operation of Cache-Based Multiprocessors," in *ISCA*, 1987, pp. 234–243.

[10] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *ISCA*, 2011, pp. 93–104.

[11] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *HPCA*, 2011, pp. 169–180.

[12] A. Gupta, W.-D. Weber, and T. C. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," in *ICPP (1)*, 1990, pp. 312–321.

[13] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[14] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *HPCA*, 2012, pp. 129–140.

[15] D. A. Wallach, "PHD: A Hierarchical Cache Coherent Protocol," Ph.D. dissertation, 1992.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," in *ISCA*, 1990, pp. 15–26.

[17] T. J. Ashby, P. Diaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Trans. Computers*, vol. 60, no. 4, pp. 472–483, 2011.

[18] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *PACT*, 2011, pp. 155–166.

[19] C. Fensch and M. Cintra, "An OS-based alternative to full hardware coherence on tiled CMPs," in *HPCA*, 2008, pp. 355–366.

[20] S. Kaxiras and G. Keramidas, "SARC Coherence: Scaling Directory Cache Coherence in Performance and Power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, 2010.

[21] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *PACT*, 2012, pp. 241–252.

[22] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: efficient hardware support for disciplined non-determinism," in *ASPLOS*, 2013, pp. 13–26.

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

[24] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *HPCA*, 2014, pp. 165–176.

[25] S. V. Adve, "Designing Memory Consistency Models For Shared-Memory Multiprocessors," Ph.D. dissertation, 1993.

[26] S. V. Adve and M. D. Hill, "Weak Ordering - A New Definition," in *ISCA*, 1990, pp. 2–14.

[27] K. Gharachorloo, "Memory Consistency Models For Shared-Memory Multiprocessors," Tech. Rep. CSL-TR-95-685, 1995.

[28] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[29] S. V. Adve and M. D. Hill, "A Unified Formalization of Four Shared-Memory Models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 6, pp. 613–624, 1993.

[30] P. J. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *ISCA*, 1992, pp. 13–21.

[31] Advanced Micro Devices, Inc., *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, May 2013, revision 3.20.

[32] R. H. B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," in *Workshop on Parallel and Distributed Debugging*, 1993, pp. 1–11.

[33] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS*, 2002, pp. 211–222.

[34] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[35] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *ISPASS*, 2009, pp. 33–42.

[36] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *PACT*, 2008, pp. 72–81.

[37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*, 1995, pp. 24–36.

[38] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IISWC*, 2008, pp. 35–46.

[39] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: streamlining STM by abolishing ownership records," in *PPOPP*, 2010, pp. 67–78.

[40] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[41] R. Rajwar and J. R. Goodman, "Speculative lock elision: enabling highly concurrent multithreaded execution," in *ISCA*, 2001, pp. 294–305.

[42] N. Barrow-Williams, C. Fensch, and S. W. Moore, "A communication characterisation of Splash-2 and Parsec," in *IISWC*, 2009, pp. 86–97.

[43] Intel Corporation, *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, 2002.

[44] ARM Limited, *ARMv8-A Reference Manual*, 2014.

[45] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, February 2014.

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

[47] A. Ros and A. Jimborean, "A Dual-Consistency Cache Coherence Protocol," in *IPDPS*, 2015.

[48] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for GPU architectures," in *HPCA*, 2013, pp. 578–590.

[49] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "QuickRelease: A Throughput Oriented Approach to Release Consistency on GPUs," in *HPCA*, 2014.

[50] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *MICRO*, 2009, pp. 423–434.