Appears in the 10th Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD 2012)
Held in conjunction with the 39th International Symposium on Computer Architecture (ISCA-39)
Portland, OR June 2012

# The Limits of Concurrency in Cache Coherence

Blake A. Hechtman, Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University
Durham, NC, USA
bah13@duke.edu, sorin@ee.duke.edu

## Abstract

*Prior work has shown how to improve the performance of cache coherence protocols by using logical time to enable concurrency that would not be legal in physical time. We hypothesized that extending this prior work to enable even greater concurrency would further improve performance, and we developed two novel techniques that leverage logical time to increase concurrency. Both schemes appear to offer significant benefits in concurrency, yet the primary result of this paper is negative. The potential benefits are clear, but common software idioms tend not to be able to exploit this potential. This negative result contributes insight into the limits of concurrency in cache coherence and informs the research community about an avenue of research that appears promising but is unlikely to yield significant gains.*

## 1 Introduction

Many shared memory systems—including both multicore processors and multi-chip multiprocessors—provide cache coherence. A cache coherence protocol ensures that the contents of the various caches are kept coherent and that cores that access these caches obtain up-to-date values of cached data. Typically, cache coherence protocols maintain the single-writer, multiple-reader (SWMR) invariant, which requires that, for a given datum (generally a cache block) at any given time, that datum is either cached in a write-able (and read-able) state by one cache or it is cached in a read-only state by zero or more caches. Thus the lifetime of a datum can be divided into a sequence of epochs, where each epoch is either read-write for one cache or read-only for zero or more caches. Coherence constrains concurrency by prohibiting, for each block, the concurrent existence of a writer and a reader or two writers.

One strategy for increasing concurrency and potentially improving the performance of a cache coherence protocol is to relax the SWMR invariant when violations of it cannot be observed. In this paper, we consider only non-speculative relaxation of the SWMR invariant, and the key to this non-speculative relaxation is to enforce SWMR in *logical time* [3] instead of physical time. Logical time is a basis of time that respects causality. That is, if event A causes event B (e.g., the sending of a message causes the reception of that message), then event A is before event B in logical time (e.g., the sending of the message is before the reception in logical time). Events that are not causally related, either directly or transitively, may be ordered in any way in logical time. Enforcing SWMR in logical time is sufficient, as we explain in Section 2, and it exposes opportunities for concurrency by sometimes allowing SWMR to be violated in physical time. Thus, even if one cache has a read-write copy of a block at the same *physical* time at which another cache has a read-only copy of the same block, this situation satisfies coherence as long as those two coherence epochs do not overlap in *logical* time. Such a system enables reads and writes to the same block that are concurrent in physical time.

Some prior systems and research proposals have exploited the opportunities provided by logical time cache coherence, but only to a fairly modest extent. In Section 3, we describe prior schemes for exploiting logical time cache coherence to enhance concurrency. In this work, we push the use of logical time significantly beyond what has been done and proposed before, and to what we believe is its farthest possible extreme. In Sections 4-6, we explain how we achieve more potential concurrency than in prior schemes.

Despite the potential benefits of extending coherence concurrency, the primary result of this research is negative: as we show in Section 7, providing more opportunities for concurrent reads and writes to the same block does not lead to significantly greater performance. Because this negative result is surprising and counter-intuitive at first—many readers may find it intuitive after the fact—we believe there is merit in sharing this result and the insights that explain it. Furthermore, in an era in which many researchers are trying to extend coherence to systems with large numbers of cores, it is beneficial to share with this community the experiences of traveling down one seemingly appealing, yet ultimately unrewarding, avenue.

## 2 Why Logical Time Coherence is Correct

It is not immediately obvious that enforcing cache coherence's SWMR invariant in logical time, instead of in physical time, is correct. By "correct", we mean that the

*Initially, block B has value zero in memory and in Core C1's cache.*

*Core C1 performs three loads to B and Core C2 performs one store to B with value 1*

| Physical time execution | | |
|---|---|---|
| Physical time | Core C1 | Core C2 |
| initially | B is read-only | B is invalid |
| 1 | Perform load #1: B=0 | Issue request for read-write |
| 2 | Receive Invalidation from C2, send Ack | |
| 3 | | Receive Ack from C1, perform store B=1 |
| 4 | Perform load #2: B=0 | |
| 5 | Perform load #3: B=0 | |
| 6 | Cache miss – issue coherence request | |

| Logical time execution | | |
|---|---|---|
| Logical time | Core C1 | Core C2 |
| Initially | B is read-only | B is invalid |
| 1 | Perform load #1: B=0 | Issue request for read-write |
| 2 | Perform load #2: B=0 | |
| 3 | Perform load #3: B=0 | |
| 4 | Receive Invalidation from C2, send Ack | |
| 5 | | Receive Ack from C1, perform store B=1 |
| 6 | Cache miss – issue coherence request | |

Fig. 1. Example execution with Scheurich's Optimization.

system satisfies its architectural specification. To understand why logical time coherence is sufficient, we must first define correct behavior. In this case of the shared memory system, architectural correctness is defined by the architecture's memory consistency model [13]. Although a thorough discussion of consistency models is beyond the scope of this paper, the key idea is that the consistency model specifies the legal orderings of loads and stores performed by multiple threads that share an address space. Sequential consistency [4], for example, specifies that the system must appear to perform all loads and stores in a total order that respects the program order at each thread.

To reason about the correctness of coherence, it is necessary to consider the relationship between coherence and consistency. It has been shown that, for many consistency models, a system can provide the consistency model by providing (physical time) cache coherence (SWMR) and some allowable reordering of loads and stores between when the core commits them and when they are applied to the memory system [8]. Sequential consistency [4] can be provided by a system with cache coherence and no reordering. SPARC's Total Store Order (TSO) [14] and the x86 consistency model [9] can be provided by a system with cache coherence and a FIFO write buffer between each core and the memory system. Weak consistency models, like Alpha [12] and ARM [1], permit even more reordering between the cores and the memory system. The majority of current systems maintain this relationship between coherence and consistency; that is, they maintain consistency by providing coherence and some reordering between the cores and the memory system. Thus, in such systems, providing coherence in physical time suffices to provide memory consistency and is thus correct.

For similar reasons, providing coherence in logical time also suffices, so long as there is a single logical time basis across all addresses. Because memory consistency models apply to all addresses, the logical time basis must pertain to all addresses. Thus a scheme that uses different logical time bases for enforcing coherence for different addresses could lead to violations of some consistency models (including sequential consistency [4] and TSO/x86 [11]).

## 3 Previous Uses of Logical Time Coherence

Escaping the constraints of implementing cache coherence in physical time reveals opportunities to increase concurrency that would not otherwise be available. In this section, we present two previous schemes for exploiting logical time coherence.

### 3.1 Scheurich's Optimization: Delayed *Processing* of Invalidations

One well-known application of logical time is an optimization developed by Scheurich and Dubios [10]. In a directory cache coherence protocol, Scheurich and Dubois showed that it is legal for a core to perform loads to an invalidated cache block until the core's next cache miss. For example, consider a core C1 that holds block B in a read-only state and then receives an invalidation for B. C1 may acknowledge the invalidation and continue to load B until C1 makes a coherence request in response to a cache miss. Until that time, C1's loads of B are logically before C1 received the invalidation and acknowledged it. We illustrate an example in Fig. 1, highlighting the loads and stores.

Scheurich's Optimization (SO), as shown in Fig. 1, allows core C1 to continue reading block B after (in *physical* time)

C1 has acknowledged invalidating B and after C2 has been granted access to change B's value from the value that C1 is still reading. This optimization has clear potential to improve concurrency. One important limitation of SO is that it does not apply to locks or flags. That is, using SO to defer an invalidation to a lock or flag does not help performance and, in fact, can hurt performance and potentially lead to deadlock.

## 3.2 Tear-Off Blocks

A subsequent scheme to SO, called tear-off blocks [5], extends physical time incoherence even farther than SO. In a directory protocol, a cache can request a tear-off block from the directory, which means that the cache informs the directory that the cache will self-invalidate the block before issuing its next coherence request (the same constraint as SO). The tear-off block can be incoherent in physical time until it is self-invalidated. In architectures that maintain sequential consistency, the tear-off block scheme is limited to having at most one tear-off block incoherent per cache. With more relaxed consistency models, a cache can have more simultaneous tear-off blocks.

## 3.3 Preliminary Conclusion

Given what we have shown thus far, it appears that there are opportunities to improve concurrency and thus performance, by exploiting logical time. We thus decided to push the use of logical time coherence to its extremes, in order to discover how much more benefit we could achieve.

## 4 Pushing Logical Time Cache Coherence to the Limits

Based on Scheurich's optimization, we made two discoveries that spurred our research:

1. We discovered that SO is overly conservative and the same insight could enable even greater concurrency.

2. We discovered that SO has a "converse." SO enables a core to continue to read a block after another core obtains read-write access to it. Our anti-SO scheme enables a core to obtain read-write access to a block before invalidating other cores with read-only access to that block.

We next describe our schemes for exploiting these observations.

| *Initially, block B has value zero in memory and in Core C1's cache.* | | | | | |
|---|---|---|---|---|---|
| *Core C1 performs four loads to B then one to A. Core C2 performs one store to B with value 1* | | | | | |
| **Physical time execution** | | | **Logical time execution** | | |
| **Physical time** | **Core C1** | **Core C2** | **Logical time** | **Core C1** | **Core C2** |
| Initially | B is read-only | B is invalid | initially | B is read-only | B is invalid |
| 1 | Perform load #1: B=0 | Issue request for read-write for B | 1 | Perform load #1: B=0 | Issue request for read-write for B |
| 2 | Receive Invalidation from C2, send Ack | | 2 | Perform load #2: B=0 | |
| 3 | | Receive Ack for B from C1, perform store B=1 | 3 | Perform load #3: B=0 | |
| 4 | Perform load #2: B=0 | | 4 | Perform load #4: B=0 | |
| 5 | Perform load #3: B=0 | | 5 | | Receive Ack for B from C1, perform store B=1 |
| 6 | Cache miss – issue coherence request for read-only for A | | 6 | Receive Invalidation from C2, send Ack | |
| 7 | Perform load #4: B=0 | | 7 | Cache miss – issue coherence request for read-only for A | |
| 8 | Receive data for A, Load A=0 | | | Receive data for A, Load A=0 | |

Fig. 2. Example execution with Extended SO.

## 5 Extended SO: Longer Delaying of Processing Invalidations

As originally proposed, SO is overly conservative and does not enable as much concurrency as it could. SO requires a core to cease loading from an invalidated block once that core issues a coherence request triggered by a cache miss. The intuition for this restriction is that the core can "pretend" that the loads occurred earlier in physical time (i.e., they do occur earlier in logical time) until the core requests something from the "outside world." That is, until the core interacts with the outside world (other cores), the outside world cannot distinguish whether the loads happened before or after the invalidation was received.

This restriction, however intuitive it may be, is more restrictive than necessary. Simply *issuing* a coherence request does not impact causality, because a request does not impact the system until it is ordered by the protocol. In a directory protocol, a request is ordered when it reaches the directory; in a snooping protocol, a request is ordered when it is serialized on the bus. Until the request is ordered, the core that issued it can continue to "pretend" that it has not occurred yet. In a directory protocol, it is difficult to leverage this observation, because few directory protocols notify a core when its request is ordered. However, in a snooping protocol, a core can extend SO until it observes its own request on the bus.

Initially, we believed that we could push SO no further. However, we then performed a thought experiment about when it is truly possible to observe a violation of causality. It turns out that it is possible to observe a violation only if a load or store instruction is committed with a value that violates causality. This observation allows a core to continue reading an invalidated block for an even longer time than is permitted by SO. The invalidation needs to be processed (i.e., the block may no longer be read) only when a load or store commits with data that was obtained at a later logical time than the logical time of the invalidation. In Fig. 2, we illustrate an example of how the invalidation can be delayed in an out-of-order core in a system with a directory cache coherence protocol. This example extends the example of SO in Fig. 1. In the example, Core C1 can continue to load block B after it issues its coherence request for block A. Note that this extended version of SO is most useful for out-of-order cores, because it enables a core to issue a coherence request for one block before invalidating the block with the deferred invalidation. This same extension applies to dynamic self-invalidation to allow for more tear-off blocks in a system that obeys sequential consistency.

## 6 Anti-SO: Delayed *Sending* of Invalidations

Scheurich's optimization showed how it is possible to delay the processing of an incoming invalidation. A core may continue to read a block after, in physical time, another core has obtained read-write access to that block. Inspired by a hunch that this optimization should be "symmetric", we have discovered a converse of SO that enables a core to delay sending invalidations while still enforcing cache coherence in logical time. That is, a core may write to a block before invalidating cores that have read-only access to that block.

### 6.1.1 High-Level Overview

Consider a core in a typical directory protocol with the four stable coherence states MOSI. It can write to a block if the block is in state M, and it can read a block if the block is in M, O, or S. If a core wishes to read or write to a block for which it has insufficient coherence permissions, it must issue a coherence request to the directory to obtain the appropriate permissions. We call these coherence requests GetShared (GetS) and GetModified (GetM). The typical directory protocol maintains the coherence invariant that, at any given physical time, there is either one writer or zero or more readers; there is no time when there is simultaneously both a writer and a reader.

We developed Anti-SO as a modification of a typical directory protocol. The key innovation of Anti-SO is that it allows a core to write to one or more blocks in its cache before sending invalidations to other cores that could have the block in a valid state. When a core wishes to write to a block for which it does not currently have read-write permissions, it begins a transaction.[1] We consider the core (not the block) to have changed its state from Normal to InTransaction. During a transaction, all of the memory accesses this core performs, both loads and stores, are logically delayed until the transaction completes. The transaction continues until an event occurs that forces the transaction to commit, such as a coherence request for Modified access to a block accessed during the transaction or the cache wishing to replace a block written during a transaction. At this point, the core commits its transaction. During a transaction, the system is non-speculatively incoherent in physical time, and committing a transaction restores the system to physical time coherence.

Starting with a baseline MOSI directory protocol, Anti-SO involves adding a few extra stable coherence states to distinguish blocks that have transactional coherence permissions. The complete list of stable states is in . We add states TS, TO, and TM, which denote that a block's state is transactionally S, O, or M, respectively.

These states are similar but not identical to their non-transactional counterparts, and we will highlight the

[1] An Anti-SO transaction differs from a transaction in transactional memory; in Anti-SO, transaction boundaries are determined by the hardware, rather than specified by the software.

differences between them in the rest of this section. The key to understand for now is that a block can be simultaneously (in physical time) TM in one cache (read-write) and either O or S (read-only) in one or more other caches. Multiple cores may be in transactions at the same time, but only one core can have a given block in state TM at a time.

### 6.1.2 Beginning a Transaction

A core begins a transaction in response to wishing to write to a block for which it does not have appropriate coherence permissions. The core enters state InTransaction and issues a coherence request to the directory for Transactionally Modified (TM) permission for the desired block. We call this request a GetTM. The directory receives this GetTM request and responds to the requesting core that it can enter state TM for this block. The core that issued the GetTM request receives the response from the directory and sets the block's state to TM. In TM, the core may read and write the block.

Unlike in a typical directory protocol, when the directory receives the GetTM, it does *not* send invalidation messages to cores that have this block in state S or O. Any cores with the block in state S or O are allowed to stay in their current state and continue reading the block. If a core has the block in state M, the directory sends it a message to change its state to O.

### 6.1.3 During a Transaction

A core's InTransaction behavior is quite different from its Normal behavior.

*Table 1. The owner of a block is the entity responsible for responding to a coherence request for that block. Shaded entries are impossible.*

| State | Normal Permission | InTransaction Permission | Owner | Compatible with cores in other states |
|---|---|---|---|---|
| I | None | None | N | All |
| S | Read-only | None | N | S, TS |
| O | Read-only | None | Y | S, TS, I |
| M | Read-write | Read-write | Y | I |
| TS | Read-only | Read-only | N | S, TS, I |
| TO | Read-only | Read-only | Y[a] | S, TS, I |
| TM | | Read-write | Y[b,c] | O, S, I |
| M/TM | Read-write | Read-write | Y[c] | I |

[a] Responds to GetS. If GetM, will commit block and
[b] Will not respond to GetS. GetS will be satisfied by cache in O (if exists) or else memory.
[c] Will commit block and go to O before responding to request.

**Performing Loads and Stores**

While in a transaction, a core can load and store to blocks in state TM, and we summarize how this works in  . A store to a block not in state TM requires the core to issue a GetTM to get TM permissions to the block. If the block is already in state M, this GetTM can be silent (i.e., not require a message to be sent to the directory) and the block changes state to a state that is both M and TM and that we denote M/TM.

A load to a block not already in a transactional state requires the core to send a request to the directory and get an acknowledgment of its transactional state. A load to a block in I, S, or O requires the core to send a GetTS, and the block ends up in either TS (if in I or S) or TO (if in O). A load to a block in M causes a silent transition to M/TM. A core can load blocks in state TS, TO, or TM. If the core later wishes to store to a block in TS or TO, it must issue a GetTM to the directory.

**Responding to Coherence Requests**

During a transaction, a core may receive coherence requests forwarded to it from the directory. The core handles the requests as shown in context switches *and page remapping will force a transaction to begin a Commit.* Requests for blocks not in TM, M/TM, or TO are handled normally, with one important exception. Consider a core C1 that has a block in state M, and receives a GetTM request from core C2. In a typical directory protocol, a request for read/write permissions would cause C1 to change its state from M to I. In Anti-SO, however, C1 changes its state from M to O. C1 remains the owner of the block and may continue to read the block. The directory will continue to forward coherence requests for normal read-only access to the block (GetS) to C1, but the directory will forward requests for transactional read and read-write access (GetTS and GetTM) to C2. This highlights the most important difference between Anti-SO and normal coherence protocols: with Anti-SO, a requestor can obtain read-write access (in TM) while another core retains read-only access.

*Table 2. Performing loads and stores. Italicized entries are the same as in the baseline directory protocol. Shaded entries are impossible.*

| State | Normal (not InTransaction) | | InTransaction | |
|---|---|---|---|---|
| | Load | Store | Load | Store |
| I | *Issue GetS → S* | Issue GetTM → TM | Issue GetS → TS | Issue GetTM → TM |
| S | *Hit* | Issue GetTM → TM | Issue GetS → TS | Issue GetTM → TM |
| O | *Hit* | Issue GetTM → TM | Issue GetS → TO | Issue GetTM → TM |
| M | *Hit* | *Hit* | Hit → M/TM | Hit → M/TM |
| TS | Hit | Issue GetTM → TM | Hit | Issue GetTM → TM |
| TO | Hit | Issue GetTM → TM | Hit | Issue GetTM → TM |
| TM | | | Hit | Hit |
| M/TM | Hit → M | Hit → M | Hit | Hit |

*Table 3. Responding to coherence requests. Shaded entries are impossible.*

| *State* | *GetS* | *GetTS* | *GetTM* | *Invalidation* |
|---|---|---|---|---|
| *I* | *Send nack*[a] | ▨ | ▨ | *Send ack → I* |
| *S* | ▨ | ▨ | ▨ | *Send ack → I* |
| *O* | *Send data* | *Send data* | *Send data* *[do not go to I]* | *Send ack → I* |
| *M* | *Send data → O* | *Send data → O* | *Send data → O* *[do not go to I]* | ▨ |
| *TS* | ▨ | ▨ | *Commit transaction → S* | *Commit transaction → S* |
| *TO* | *Send data* | *Send data* | *Commit transaction → O* | *Commit transaction → O* |
| *TM* | *Send nack*[a] | *Commit transaction → M* | *Commit transaction → M* | ▨ |
| *M/TM* | *Commit transaction → O* | *Commit transaction → O* | *Commit transaction → O* | ▨ |

[a] Both of these situations involve complicated races of messages, and nacks are the simplest approach to dealing with them. Other solutions may exist, but optimizing for these rare race cases is not worthwhile.

**Events that Force an Entire Transaction to be Committed**

There are three events that force a core to commit a transaction:

- The core wants to evict a block in state TM, M/TM, TO, or TS,
- The core receives a forwarded GetTM for a block in TM, M/TM, TO, or TS, or
- The core receives a forwarded GetTS for a block in state TM or M/TM.

If any of these events occurs, the core starts a Commit of all TM blocks, described in Section 6.1.4. Blocks in TS, TO, and M/TM can remain in those states. In addition, context switches and page remapping will force a transaction to begin a Commit.

### 6.1.4 Committing a Transaction

To commit a transaction, the core must atomically change the state of all TM blocks to M. During this process, the core enters state InCommit and delays responding to forwarded coherence requests for TM blocks. To commit the transaction, the core issues Commit coherence requests to the directory for every block in state TM. These Commit requests are equivalent to requests for Modified (state M) permissions, and they invalidate the copies of the blocks in all other caches. When the committing core receives an acknowledgment of a Commit for a block, it changes that block's state to M. Once the committing core receives acknowledgments for all of its Commits, the transaction is complete and the core changes its state from InCommit to Normal.

## 6.2 An Example of Anti-SO in Action

We now walk through an example of Anti-SO in action. As illustrated in Fig. 3, in a system with Anti-SO, C1 obtains TM access to A and enters InTransaction. It stores to A and then requests and obtains TM access to B before storing to B. Note that C1 has performed stores to two blocks that are still readable by C2; this situation violates physical time coherence. Assume for now that C1 decides to complete its transaction at this point (e.g., because it wishes to evict A from its cache). C1 commits this transaction and logically inserts the transaction (i.e., the stores in the transaction) into the total order required by SC *after* the loads by C2. As part of committing the transaction, C1 must change the state of all

| | | | |
|---|---|---|---|
| Initially, Mem[A] = Mem[B] = 0 | | | |
| *Core C1 performs one Store to A then B, both with value 1. Core C2 loads A then loads B.* | | | |
| | | | |
| Cycle | Core C1 | Directory | Core C2 |
| 0 | Begin transaction; change core state from Normal to InTransaction; issue GetTM request for A | | Load r1, A // r1 = 0 |
| 10 | | Receive GetTM for A | Load r2, B // r2 = 0 |
| 20 | Obtain TM permission for A; Store A, 1 | | |
| 30 | Issue GetTM request for B | | |
| 40 | | Receive GetTM for B | |
| 50 | Obtain TM permission for B; Store B, 2; Start to commit transaction; change core state to InCommit; send Commit request for A and B to directory | | |
| | | Receive Commits | |
| | | | Receive invalidations for A, B |
| 80 | Obtain M permissions for A and B; change core state to Normal | | |

Fig. 3. Example execution with Anti-SO

blocks accessed in the transaction from TM to M by invalidating A and B from C2's cache. If a core wishes to evict a block in TS, TO, or M/TM—which can happen either during InTransaction mode or in Normal mode—the core needs to commit just that block if in Normal mode or all blocks in a transaction if in InTransaction mode. To commit a block, the core issues a Commit coherence request for that block to the directory, and the directory responds with an acknowledgment. Upon receiving the ack, the block's state changes from TS to S, TO to O, and M/TM to O.

This single, simple example illustrates at a high level how Anti-SO handles one situation and hopefully provides some intuition for how it improves performance through temporary incoherence.

## 7 The Negative Result

Our hypothesis was that, if Scheurich's optimization and other logical time coherence "tricks" have been successfully employed, then there should be merit in pushing logical time coherence even farther. We implemented both extended SO and Anti-SO in a full-system simulation environment (Simics [6] and GEMS [7]), and we compared them to each other and to a baseline system without either optimization. For both SO and Anti-SO, we initially made optimistic assumptions about their implementations, to get a better idea of their ceilings before focusing on more realistic implementations. For example, for Anti-SO, we ignored the complexity and latency of finding all blocks that need to be committed at the end of a transaction.

The results of the experiments were surprising and disappointing: the differences in performance were less than the error margins in our results. (Thus, we present no graphs.) As with many discoveries, the intuition for our negative result appears far clearer in retrospect. In this section, we explain why pushing logical time coherence farther does not help performance for common software idioms, and we show a few scenarios in which it does happen to help.

### 7.1 Common Software Idioms

Logical time cache coherence enables concurrent, *conflicting* accesses to shared data, where accesses are said to conflict if they are to the same address and at least one is a write. The catch, though, is that most programs do not benefit from concurrent, conflicting accesses to shared data. Concurrent conflicting accesses, by definition, are part of a race. In data-race-free (DRF) programs, the only accesses that race are accesses to synchronization variables such as locks. By using synchronization accesses to enforce critical sections, a DRF program avoids races for non-synchronization data. Thus permitting concurrent, conflicting accesses to non-synchronization data offers little advantage. There will not be situations in which an invalidation will arrive for data necessary to complete a critical section; thus Scheurich's optimization does not help. Similarly, there will not be situations in which obtaining read-write permission would invalidate data from another thread that still needs to load that data during a critical section; thus, anti-SO does not help.

For accesses to synchronization variables, the optimal situation is to enforce physical time coherence, because physical time coherence minimizes the time between a lock release and when the lock can be acquired by another thread. The goal is to make the changes to synchronization variables visible as soon as possible. By contrast, logical time coherence permits later updates of synchronization variables, which is not helpful.

Intuitively, DRF programs enforce causality—the basis of logical time—and thus there is little difference between physical time and logical time.

### 7.2 Benefits in Uncommon Cases

There are two situations in which logical time coherence offers potential benefits: false sharing and prefetching.

**False Sharing.** When a block holds multiple pieces of data, it is possible for false sharing to occur. Because the sharing is false, there is not necessarily a causal relationship between accesses to the pieces of data in the block. This is an opportunity to use logical time coherence because there is a difference between physical time and logical time. In physical time, the data is shared; in logical time, it is not shared. Logical time schemes, like SO and anti-SO, can mitigate the impact of false sharing by allowing a thread to concurrently read from one part of a block while another thread is writing to another part of the block. These concurrent, conflicting accesses do not violate causality. However, because logical time coherence does not allow concurrent writes, it is not a complete solution to false sharing. Furthermore, previous schemes to handle false sharing exist, including sub-block coherence [2].

**Prefetching.** Consider a thread T1 that is executing within a critical section. In a DRF program, it should not receive an invalidation for any block in the critical section. However, with prefetching, it is possible for another thread T2's prefetch for read-write permissions to cause an invalidation to arrive at T1 during the critical section. SO and Anti-SO address this problem by allowing T1 to continue to read the block if necessary. If T2's prefetch was early (i.e., issued before T2 reached its critical section) or if the prefetch ends up not being used (e.g., due to a mis-prediction or an overly aggressive prefetcher), then SO and Anti-SO help. However, SO and Anti-SO do not address the problem of a prefetch arriving at a thread that needs to continue writing (rather than reading) a block. Furthermore, there are many other schemes for mitigating the impacts of overly aggressive prefetching.

## 8 Conclusions

The performance of cache coherence protocols is becoming increasingly important as commodity processor chips continue to incorporate more cores and to provide cache-coherent shared memory. One avenue of research has previously explored the potential to improve coherence performance by using logical time reasoning to increase concurrency. Based on the promise of this research and the use of logical time techniques in actual systems, we sought to extend this approach to its limits. In the course of doing so, we discovered a negative result, except for the potential to use logical time coherence in specific situations (false sharing and prefetching).

**Acknowledgments**

This work has been supported in part by the Semiconductor Research Corporation under contract 2009-HJ-1881. We thank Jacob Harer and Ralph Nathan for early contributions to what became the Anti-SO protocol. We thank Alvy Lebeck for his helpful feedback on Anti-SO and on this paper.

**References**

1. ARM. *ARM v7A+R Architectural Reference Manual*. 2008.
2. Kadiyala, M. and Bhuyan, L.N. A Dynamic Cache Sub-block Design to Reduce False Sharing. *Proceedings of the 1995 International Conference on Computer Design*, (1995).
3. Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (1978), 558–565.
4. Lamport, L. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28*, 9 (1979), 690–691.
5. Lebeck, A.R. and Wood, D.A. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (1995), 48–59.
6. Magnusson, P.S., Christensson, M., Eskilson, J., et al. Simics: A Full System Simulation Platform. *IEEE Computer 35*, 2 (2002), 50–58.
7. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News 33*, 4 (2005), 92–99.
8. Meixner, A. and Sorin, D.J. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. *IEEE Transactions on Dependable and Secure Computing*, (2009).
9. Owens, S., Sarkar, S., and Sewell, P. A Better x86 Memory Model: x86-TSO. *Proceedings of the Conference on Theorem Proving in Higher Order Logics*, (2009).
10. Scheurich, C. and Dubois, M. Correct Memory Operation of Cache-Based Multiprocessors. *Proceedings of the 14th Annual International Symposium on Computer Architecture*, (1987), 234–243.
11. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., and Myreen, M.O. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM*, (2010).
12. Sites, R.L., ed. *Alpha Architecture Reference Manual*. Digital Press, 1992.
13. Sorin, D.J., Hill, M.D., and Wood, D.A. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.
14. Weaver, D.L. and Germond, T., eds. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.