OSPREY: Implementation of Memory Consistency Models for Cache Coherence Protocols involving Invalidation-Free Data Access

George Kurian†, Qingchuan Shi‡, Srinivas Devadas†, Omer Khan‡

† Massachusetts Institute of Technology – {gkurian, devadas}@csail.mit.edu
‡ University of Connecticut – {qingchuan.shi, khan}@uconn.edu

***Abstract*—Data access in modern processors contributes significantly to the overall performance and energy consumption. Traditionally, data is distributed among the cores through an on-chip cache hierarchy, and each producer/consumer accesses data through its private level-1 cache relying on the cache coherence protocol for consistency. Recently, remote access, a mechanism that reduces energy and latency through word-level access to data anywhere on chip has been proposed. Remote access does not replicate data in the private caches, and thereby removes the need for expensive cache line invalidations or updates. Researchers have implemented remote access as an auxiliary mechanism in cache coherence to improve efficiency. Unfortunately, stronger memory models, such as Intel's TSO, require strict ordering among the loads and stores. This introduces serialization penalties for data classified to be accessed remotely, which hampers each core's ability to optimally exploit memory level parallelism.**

In this paper we propose a novel timestamp-based scheme to detect memory consistency violations. The proposed scheme enables remote accesses to be issued and completed in parallel while continuously detecting whether any ordering violations have occurred, and rolling back the pipeline state (if needed). We implement our scheme for the locality-aware cache coherence protocol that uses remote access as an auxiliary mechanism for efficient data access. Our evaluation using a 64-core multicore processor with out-of-order speculative cores shows that the proposed technique improves completion time by 26% and energy by 20% over a state-of-the-art cache management scheme.

I. Introduction

Increasing the number of cores has replaced clock frequency scaling as the method to improve performance in modern multicore processors. These multiple cores can either be used in parallel by multiple applications or by multiple threads of the same application to complete work faster. Maintaining good single-core performance and multicore scalability are of the utmost importance in continuing to improve performance and energy efficiency.

Since the working sets of applications rarely fit within the L1 cache, ensuring good single-core performance requires the exploitation of the memory level parallelism (MLP) in an application. MLP can be exploited using out-of-order (OOO) cores through dynamically scheduling independent memory operations. In addition to being ubiquitous in server processors, current industry trends are moving towards OOO cores in embedded (Atom [1], ARM [2]), and energy-efficient high-performance processors (Knights Landing [3]) as well.

Maintaining high multicore scalability requires addressing the challenges that arise when placing multiple interacting cores on the same die. First, a higher number of cores introduces greater pressure on off-chip memory to satisfy the needs of all the cores. Second, a higher core count increases the sensitivity to network bandwidth, latency and energy efficiency since an increasingly higher fraction of memory requests is spent in the network.

State-of-the-art multicore processors must also balance ease of programming with good performance and energy efficiency. The programming complexity is significantly affected by the memory consistency model of the processor. The memory model dictates the *order* in which the memory operations of one thread appear to another. The strongest memory model is the Sequential Consistency (SC) [4] model. SC mandates that the global memory order is an interleaving of the memory accesses of each thread with each thread's memory accesses appearing in program order in this global order. SC is the most intuitive model to the software developer and is the easiest to program and debug with.

Commercial processors do not implement SC due to its negative performance impact. ARM [2] and IBM Power [5] processors implement relaxed (/weaker) memory models that allow reordering of load and store instructions with explicit fences for ordering when needed. These processors can better exploit MLP, but require careful programmer-directed insertion of memory fences to do so. Automated fence insertion techniques sacrifice performance for programmability [6].

Intel x86 [7] and SPARC [8] processors implement Total Store Order (TSO), which attempts to strike a balance between programmability and performance. The TSO model only relaxes the Store→Load ordering of SC, and improves performance by enabling loads (that are crucial to performance) to bypass stores in the write buffer. Note that fences may still be needed in critical sections of code where the Store→Load ordering is required.

Implementing the TSO model on OOO core based processors in a straightforward manner sacrifices memory level parallelism. This is because loads have to wait for all previous load/fence operations to complete before being issued while stores/fences have to wait for all previous load/store/fence operations. This inefficiency is circumvented in current processors by employing two optimizations [9]. (1) Load performance is improved using speculative execution, enabling loads to be issued and completed before previous load/fence operations. Memory consistency violations are detected when invalidations, updates or evictions are made to addresses in the load queue. The pipeline state is rolled back if this situation arises. (2) Store performance is improved using exclusive store prefetch requests. These prefetch requests fetch the cache line into the L1-D cache and can be executed out-of-order and in parallel. The store requests, on the other hand, must be issued and completed in-order to preserve TSO. However, most store requests hit in the L1-D cache (due to the earlier prefetch

request) and hence can be completed quickly. The performance of fences is automatically improved by optimizing the previous load and store operations. Note that the above two optimizations can also be employed to improve the performance of processors under sequential consistency or memory models weaker than TSO.

A. *Memory Consistency Models and Data Access* Efficiency

Unfortunately, cache coherence protocols that intelligently avoid cache line invalidations or updates are incompatible with these optimizations. For example, the locality-aware cache coherence protocol[1] has been recently proposed to improve on-chip memory access latency and energy efficiency in large-scale multicores [10]. This protocol is motivated by the observation that cache lines exhibit varying degrees of reuse (i.e., variable spatio-temporal locality) at the private cache levels. A cache-line level classifier is introduced to distinguish between low and high-reuse cache lines. A traditional cache coherence scheme that replicates data in the private caches is employed for high-reuse data. Low-reuse data is handled efficiently using a *remote access* [11] mechanism that does not allocate data in the private cache levels. Instead, it allocates only a single copy in the designated core's shared cache slice and directs load and store requests made by all other cores towards it. Data access is performed at the word level and requires a roundtrip message between the requesting core and the remote cache slice. This improves the utilization of private cache resources by removing unnecessary data replication. In addition, it reduces network traffic by transferring only those words in a cache line that are accessed on-demand. Consequently, unnecessary invalidations and write-back requests are removed that reduce network traffic even further.

A drawback of using remote access in cache coherence protocols is that invalidation/update requests are avoided and thereby, cannot be used to detect memory consistency violations for speculatively executed load operations. In addition, an exclusive store prefetch request is not applicable since a remote access never caches data in the private cache. We seek to develop a new micro-architectural mechanism that retains the advantages of auxiliary techniques, such as remote access, for efficient data access, and at the same time removes their dependence on the underlying cache coherence protocol for memory consistency violation detection.

B. *Principal Contributions of OSPREY*

We propose a novel timestamp-based scheme to detect memory consistency violations in multicores that implement speculative execution and *invalidation-free* data access protocols. To demonstrate the applicability of our proposed scheme, we extend the recently proposed locality-aware coherence protocol to work with OOO speculative cores for popular memory models. Each load and store operation is assigned an associated timestamp and a simple arithmetic check is done at commit time to ensure that memory consistency has not been violated. The timestamp mechanism is efficient due to the observation that consistency violations occur due to conflicting accesses that have temporal proximity (i.e., within a few cycles of each other), thus requiring timestamps to be stored only for a small time window. This technique works completely in hardware and requires only $2.2KB$ of storage per core. This scheme guarantees forward progress and is starvation-free.

Our evaluation using a 64-core multicore with out-of-order speculative cores shows that the *timestamp-based* memory consistency violation detection scheme, when implemented on top of the locality-aware cache coherence protocol [10], improves completion time by 26% and energy by 20% over a state-of-the-art cache management scheme (Reactive NUCA [12]).

[1]Locality-aware cache coherence protocol has been evaluated for the Sequential Consistency (SC) memory model using in-order cores with a single outstanding memory transaction per-core [10].

II. Baseline Multicore System

The baseline is a tiled multicore processor with a 2-D mesh interconnection network. Each core consists of a compute pipeline, private L1 instruction and data caches, a physically distributed shared L2 (/LLC) cache with integrated directory, and a network router. The coherence directory is integrated with the LLC slices by extending the tag arrays and tracks the sharing status of the cache lines in the per-core private L1 caches. The private L1 caches are kept coherent using the ACKwise limited directory-based coherence protocol [13]. Some cores have a connection to a memory controller as well.

The mesh network uses dimension-order X-Y routing and wormhole flow control. Reactive-NUCA's [12] data placement, replication and migration mechanisms are used to manage the LLC. Private data is placed at the L2 slice of the requesting core, shared data is address interleaved across all L2 slices, and instructions are replicated at a single L2 slice for every cluster of 4 cores using a rotational interleaving mechanism.

When a core makes a memory request that misses the private L1 cache, the locality-aware protocol [10] either brings the entire cache line using a traditional directory scheme, or just accesses the requested word at the shared L2 cache location using remote access. This decision is based on the spatio-temporal locality of the cache line. The reuse is profiled at runtime using hardware counters in the private L1 cache and the shared coherence directory. These counters are maintained at a cache line granularity. A classifier subsequently uses this profiled information to mark data as privately cacheable at the L1 cache or remotely accessed at the shared L2 cache based on a watermark called the Private Caching Threshold (PCT). The classification decision is continuously adapted at runtime so as to closely track the behavior of the application.

III. Timestamp-based Consistency Validation

This section introduces the proposed timestamp-based technique for detecting memory consistency violations. This technique allows all load/store operations to be executed speculatively. Speculation failure is detected by associating timestamps with every memory transaction and performing a simple arithmetic check at commit time. We describe the working of this technique under the popular TSO memory model. Later, in Section IV-A, we discuss how it can be extended to stronger (i.e., SC) or weaker memory models.

The *timestamp*-based technique is built up gradually through a sequence of steps. The microarchitecture changes needed are colored in orange in Figure 1, and will be described when

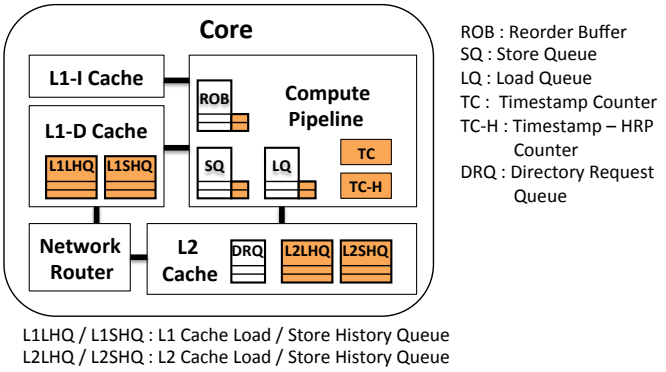

Fig. 1. Microarchitecture of a Multicore Tile. The orange-colored modules are added to implement the proposed modifications.

introduced. The implementation is first described for a pure remote access scheme (that always accesses the shared L2 cache). Later, we describe adaptations for the locality-aware protocol that combines both remote L2 and private L1 cache accesses according to the reuse characteristics of data.

A. Simple Implementation of TSO Ordering

We first introduce a straightforward method to ensure TSO ordering (without speculation). The TSO model can be implemented by enforcing the following *two* constraints: (1) Load operations wait (i.e., without being issued to the cache subsystem) till all previous loads and fences complete; (2) Store operations and fences wait till all previous loads, stores and fences complete. If all memory references are satisfied by the L1 cache, this scheme works quite well. However, a memory transaction that misses in the L1 cache takes several (∼10-100) cycles to complete. During this time, the load and store queues fill up quickly and stall the pipeline.

B. Basic Timestamp Scheme

Next, we formulate a mechanism to increase the performance of load operations (instead of making them wait as described above). This requires enabling loads to execute (speculatively) as soon as their address operand is ready, potentially out-of program order and concurrently with other loads. This could violate the TSO memory consistency model, and hence, a validation step is required to ensure that memory ordering is preserved. Stores and fences, on the other hand, have to wait till all previous loads, stores and fences have been completed.

To build a deeper understanding for the validation step, consider the example shown in Figure 2 that illustrates how it works in current out-of-order processors. Core 1 executes LOAD A followed by LOAD B, while Core 2 executes STORE B followed by STORE A. Initially, both A and B contain the value '0'. The load/store requests execute and arrive at their ordering points in the sequence shown in Figure 2. LOAD B arrives first, followed by STORE B, STORE A and finally, LOAD A. This ordering is permitted as loads can execute speculatively and out-of-order with previous loads. LOAD B returns the value '0' while LOAD A returns the value '1'. LOAD A is allowed to commit. However, LOAD B cannot commit since its execution violates TSO ordering.

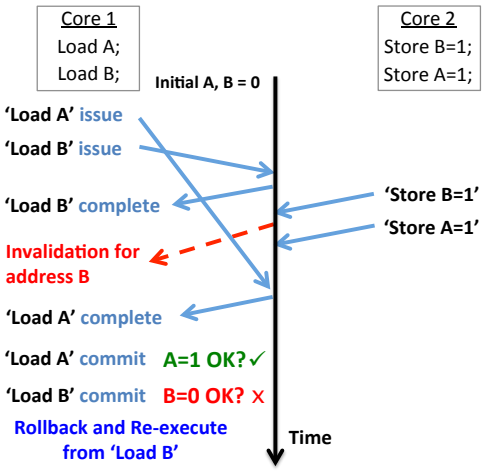

Fig. 2. Example of TSO consistency violation due to load speculation.

TSO mandates that the global memory order should respect both Load→Load and Store→Store program ordering.

TSO memory ordering can be enforced by relying on cache coherence messages and performing a consistency validation check at the commit time of each load operation. In this example, STORE B generates an invalidation request for Core 1 (as shown in Figure 2). This invalidation message checks for an outstanding load to address B in Core 1's load queue, and if present, sets the "conflict" flag. When the commit-time check for LOAD B determines that the conflict flag has been set, the core pipeline is rolled back and execution is restarted from the load operation. Note that an eviction request for address B triggers the pipeline rollback as well.

With an invalidation-free data access protocol, STORE B does not generate an invalidation request for Core 1. Hence, the previously described check cannot be used to detect memory consistency violations. The following sections describe how a timestamp-based validation technique can be used to detect violations for such protocols.

1) Timestamp Generation: Timestamps are generated using a per-core counter, the Timestamp Counter (TC), as shown in Figure 1. TC increments on every clock cycle. Assume for now that timestamps are of infinite width, i.e., they never rollover and all cores are in a single clock domain. We will remove the infinite width assumption in Section III-D and discuss the single clock domain assumption in Section IV-B. Timestamps are tracked at different points of time, e.g., during load issue, store completion, etc. and comparisons are performed on these timestamps to determine whether speculation has failed according to the algorithms discussed in the following subsections.

2) Microarchitecture Modifications: The following changes are made to the L2 cache, the load queue and the store queue to facilitate the tracking of timestamps.

L2 Cache: The shared L2 cache is augmented with an *L2 Load History Queue (L2LHQ)* and an *L2 Store History Queue (L2SHQ)* as shown in Figure 1. They track the times at which loads and stores have been performed at the L2 cache. The timestamp assigned to a load/store is the time at which the request arrives at the L2 cache. Each entry in the L2LHQ / L2SHQ has two attributes, *Address* and *Timestamp*. An entry

Fig. 3. Structure of a load queue entry.

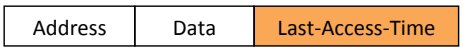

Fig. 4. Structure of a store queue entry.

is added to the L2LHQ or L2SHQ whenever a remote load or store arrives. Assume for now that the L2LHQ / L2SHQ are of infinite size. We will remove this requirement in Section III-C.

Load Queue: Each load queue entry is augmented with the attributes shown in Figure 3. The *IssueTime* field records the time at which the load was issued to the cache subsystem and the *LastStoreTime* is the most recent modification time of the cache line. A remote load obtains this timestamp from the L2SHQ. If there are multiple entries in the L2SHQ corresponding to the address, the most recent entry's timestamp is taken. This is then relayed back to the core over the on-chip network along with the word that is read.

Store Queue: Each store queue entry is augmented with the attributes shown in Figure 4. The Data contains the word to be written. The *LastAccessTime* field records the most recent access time of the cache line at the L2 cache. A remote store obtains the most recent timestamps from the L2LHQ and L2SHQ respectively. The maximum of the two timestamps is computed to get the *LastAccessTime*. This is then communicated back to the core over the on-chip network along with the acknowledgement for the store.

Each core also maintains the *OrderingTime* and *StoreOrderingTime* fields that are used to detect speculation violations. These fields are needed to precisely track the timestamp history that may otherwise be lost due to newer allocations in the load and store queues.

3) Speculation Violation Detection: In order to ensure that speculatively executed loads conform to the TSO memory consistency model, the *IssueTime* of a load must be greater than or equal to:

- The *LastStoreTime* observed by previous load operations. This ensures that the current speculatively executed load can be placed after all previous load operations in the global memory order. In other words, the global memory order for any pair of load operations from the same thread respects program order. In case the above condition is not met, the Load →Load ordering requirement of TSO is violated due to an intervening store operation.
- The *LastAccessTime* observed by previous store operations that are separated from the current load by a memory fence (MFENCE in x86). This ensures that the load can be placed after previous fences in the global memory order, i.e., the Fence → Load ordering requirement of TSO is met.

The *OrderingTime* field conveniently holds the instantaneous maximum of the above two timestamps. The functions in Algorithm 1 are responsible for updating this field and performing the necessary consistency validation check. The COMMITLOAD function is executed when a load is ready to be committed. Speculation failure is determined by comparing the *IssueTime* of the load against *OrderingTime*. If the *IssueTime* is greater, speculation has succeeded and the *OrderingTime* is updated to reflect the *LastStoreTime* observed by the load. Else, speculation has failed, and the instructions are replayed starting from the current load.

Algorithm 1 : Basic Timestamp Scheme

```
1: function COMMITLOAD()
2:     if IssueTime < OrderingTime then
3:         REPLAYINSTRUCTIONSFROMCURRLOAD()
4:         return
5:     if OrderingTime < LastStoreTime then
6:         OrderingTime ← LastStoreTime
7: function COMPLETESTORE()
8:     if StoreOrderingTime < LastAccessTime then
9:         StoreOrderingTime ← LastAccessTime
10: function COMMITFENCE()
11:     if OrderingTime < StoreOrderingTime then
12:         OrderingTime ← StoreOrderingTime
```

The COMPLETESTORE function is executed when a store completes execution, i.e., it is removed from the store queue and inserted into the cache hierarchy. Note that the store could have been committed (possibly much earlier) as soon as its address calculation and translation are done. The *StoreOrderingTime* field in the store queue keeps track of the maximum of the *LastAccessTime* observed by previously completed stores and is updated at the time of completion of each store.

The COMMITFENCE function is executed when a fence is ready to be committed. This function updates the *OrderingTime* to reflect the *StoreOrderingTime* field and serves to maintain the Fence →Load ordering.

Why does this work? The above validation check suffices to ensure that the global memory order respects program order (except for the Store→Load order). The *OrderingTime* field at the end of the COMMITLOAD and COMMITFENCE functions indicates the position of the corresponding load and fence operations in the global memory order while the *StoreOrderingTime* field at the end of the COMPLETESTORE function indicates the position of the corresponding store operation. These positions reflect the ordering of operations to the same address from multiple program threads as well.

If sequential consistency (SC) is to be implemented instead of TSO, all store operations are marked as being accompanied by an implicit memory fence. So, for speculation to be successful, the *IssueTime* of a load operation must be greater than or equal to the maximum of the *LastStoreTime* and *LastAccessTime* observed by previous load and store operations respectively.

4) Consistency Validation Example: An example illustrating the working of the timestamp-based consistency validation check is shown in Figure 5. The memory access pattern is the same as that studied previously in Figure 2. The timestamps associated with each event (such as load issue, commit, etc.) are as shown in Figure 5. Initially, assume that the *OrderingTime* at Core 1 is '3' (from previously executed loads/stores to other memory addresses). LOAD A and LOAD B are issued and completed in the order shown. Speculative execution allows LOAD B to complete before LOAD A on Core 1. When LOAD B completes (@'15'), the *LastStoreTime* is recorded as '0' since no stores have been made to ADDRESS

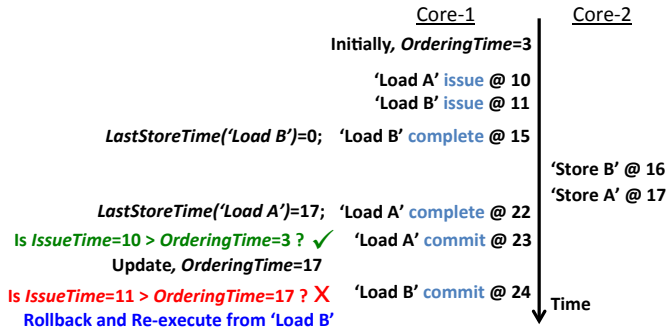

Fig. 5. Using timestamp scheme to detect violations due to speculative loads. The timestamps associated with each event are as shown. The memory access pattern is the same as shown in Figure 2.

B so far. On the other hand, when LOAD A completes (@'22'), the *LastStoreTime* is recorded as '17' since STORE A from Core 2 has been performed at time '17'.

Now, when LOAD A commits (@'23'), the COMMITLOAD function is executed which compares the *IssueTime* of LOAD A (='10') against the *OrderingTime* (='3'). Since the *IssueTime* is $\geq$ *OrderingTime*, Core 1 safely commits LOAD A. In addition, the *OrderingTime* is updated to reflect the observed *LastStoreTime* (='17'). However, when LOAD B commits (@'24'), the *IssueTime* (='11') is $<$ the *OrderingTime* (='17'). Hence, the TSO validation check fails. The pipeline state in Core 1 is rolled back and execution is restarted at LOAD B. Also, observe that when LOAD B restarts, its *IssueTime* is always greater than the *OrderingTime* (='17'), and hence it always commits (guaranteeing forward progress).

C. Finite History Queues

A drawback of the algorithm discussed so far is that the size of the load/store history queues cannot be bounded (i.e., they can grow arbitrarily large). The objective of this section is to bound the size of the L2 history queues. Note that the timestamps could still grow arbitrarily large (we will remove this requirement in Section III-D).

The history queues can be bounded by the observation that the processor cores only care about load/store request timestamps within the scope of their reorder buffer (ROB). For example, if the oldest micro-op in the ROB has been dispatched at 1000 cycles, the processor core does not care about load/store history earlier than 1000 cycles. This is because memory requests from other cores carried out before this time will not cause consistency violations. Hence, history needs to be retained only for a limited interval of time called the *History Retention Period (HRP)*. After the retention period expires, the corresponding entry can be removed from the history queues. How long should the history retention period be? Intuitively, if the retention period is equal to the maximum lifetime of a load or store operation starting from dispatch (to the reorder buffer) till completion, no usable history will be lost. However, the maximum lifetime of a memory operation cannot be bounded in a modern multicore system due to non-deterministic queuing delays in the on-chip network and memory controller.

An alternative is to set HRP such that nearly all ($\sim$99%) memory operations complete within that period. If operations do not complete within HRP, then spurious violations might occur. As long as these violations only affect overall system performance and energy by a negligible amount, they can be tolerated. Increasing the value of HRP reduces spurious violations but requires large history queues (L2LHQ/L2SHQ) while decreasing the value of HRP has the reverse effect. Finding the optimal value of HRP is critical to ensuring good performance and energy efficiency.

1) Speculation Violation Detection with Finite Queues: Speculation violations are detected using the same algorithm in Section III-B. However, since entries can be removed from the L2 history queues, checking the history queues may not yield the latest load and store timestamps. Hence, an informed assumption regarding previous history has to be made. If no load or store history is observed for a memory request, it can be safely assumed that the request did not observe any load/store operations at the L2 cache after 'CompletionTime − HRP' (CompletionTime represents the time when the load/store request completes). Hence, the *LastStoreTime* and *LastAccessTime* required by the algorithm in Section III-B are adjusted as shown by the ADJUSTHISTORY function in Algorithm 2. Note that 'NONE' indicates no load/store history for a particular address.

Algorithm 2 : Finite History Queues

```
1: function ADJUSTHISTORY()
2:     if LastLoadTime = NONE then
3:         LastLoadTime ← CompletionTime − HRP
4:     if LastStoreTime = NONE then
5:         LastStoreTime ← CompletionTime − HRP
6:     LastAccessTime ← MAX(LastLoadTime, LastStoreTime)
```

2) Finite Queue Management:: Adding entries to the history queue and searching for an address works similar to the description in Section III-B. However, with a finite number of entries, two extra considerations need to be made.

1) History queue overflow needs to be considered and accommodated.
2) Queue entries need to be pruned after the history retention period (HRP) expires.

The finite history queue is managed using a set-associative structure that is indexed based on the address (just like a regular set-associative cache). The overheads associated with the history queues are discussed in Section III-G.

Queue Insertion: When an entry (<Address, Timestamp>) needs to be added to the queue, the set corresponding to the address is first read into a temporary register. A pruning algorithm is applied on this register to remove entries that have expired (this will be explained later). Then the <Address, Timestamp> pair is added to the set as follows. If the address is already present, then the maximum of the already present timestamp and the newly added timestamp is computed and written. If the address is not present, then the algorithm checks whether an empty entry is present. If yes, then the new timestamp and address are written to the empty entry. Else, the oldest timestamp in the set is retrieved and evicted. The new <Address, Timestamp> pair is written in its place. In addition to the *set-associative structure*, each queue also contains a

Conservative Timestamp (*ConsTime*) field. The *ConsTime* field is used to hold evicted timestamps that have overflowed till they expire.

Pruning Queue: A pruning algorithm removes entries from the queue after their retention period (HRP) has expired (and/or) resets the *ConsTime* field. This function uses a second counter called TC-H (shown in Figure 1). This counter lags behind the timestamp counter (TC) by HRP cycles. If a particular timestamp is less than TC-H, the timestamp has expired and can be removed from the queue. On every processor cycle, the TC-H value is also compared to *ConsTime*. If equal, the *ConsTime* has expired and can be reset to 'NONE'.

Searching Queue: When searching the queue for an address, the set corresponding to the address is first read into a temporary register. If there is an address match, then the timestamp is returned. Else, the *ConsTime* field is returned. To maintain maximum efficiency, the *ConsTime* field should be 'NONE' (expired) most of the time, so that spurious load/store times are not returned.

For efficiency, all entries in a set are searched in parallel. However, since these history queues are really small (cf. Section III-G), the above computations can be performed within the cache access time.

D. In-Flight Transaction Timestamps

One of the main drawbacks of the algorithm presented in Section III-B was that the timestamps should be of infinite width, i.e., they are never allowed to rollover during the operation of the processor. This drawback can be removed by the observation that only the timestamps of memory operations present in the reorder buffer (ROB) need to be compared when detecting consistency violations, i.e., only memory transactions that have temporal proximity could create violations. Hence, Load and Store timestamps need to be distinct and comparable only for recent memory operations.

Finite Timestamp Width (TW): The above observation can be exploited to use a finite timestamp width (TW). When the timestamp counter reaches its maximum value, it rolls over to '0' in the next cycle. The range of possible values that the counter can take could be divided into two quantums, an 'even' and an 'odd' quantum. During the even quantum, the most significant bit (MSB) of the timestamp counter is '0' while during the odd quantum, the MSB is '1'. For example, if the timestamp width (TW) is 3 bits, values 0-3 belong to the even quantum while values 4-7 belong to the odd quantum. Now, to check which timestamp is greater than the other, it needs to be known whether the current quantum (i.e., the MSB of the TC [Timestamp Counter]) is an even or an odd quantum. If the current quantum is even, then any timestamp with an even quantum is automatically greater than a timestamp with an odd quantum. If two timestamps of the same quantum are compared, a simple arithmetic check suffices to know which is greater.

Recency of Timestamps: A possible problem with the above comparison is when the timestamps that are compared are not recent. For example, consider a system with a timestamp width (TW) of 3 bits. Assume TC is set to 3 to start with. Timestamp, T_A, is now generated and set to the value of TC, i.e., 3. Then, TC increments, reaches its maximum value of 7 and rolls over to 1. Now, another timestamp, T_B, is set to 1. If the check, $T_A > T_B$ is now performed, the result is *true* according to the algorithm discussed above. But, T_A was generated before T_B, so the result should have been *false*. The comparison check returned the wrong answer because T_A was 'too old' to be useful. Timestamps have to be 'recent' in order to return an accurate answer during comparisons. Given a particular value of the timestamp counter (TC), timestamps have to be generated in the current quantum or the previous quantum to be useful for comparison. In the worst case, a timestamp should have been generated at most 2^{TW-1} cycles before the current value of TC to be useful.

Consistency Check: In the algorithms described previously, the only arithmetic check done using timestamps is at the commit point of load operations. The check performed is: *IssueTime* $\geq$ *OrderingTime*. If both *IssueTime* and *OrderingTime* are recent, the check always returns a correct answer, else it might return an incorrect answer. Now, if *IssueTime* is recent and *OrderingTime* is old, the 'correct' answer for the consistency check is *true*, however, it might return *false* in certain cases. The answer being 'false' is OK, since all it triggers is a false positive, i.e., it triggers a consistency violation while in reality, there is no violation. As long as the number of false positives is kept low, the system functions efficiently. So, the important thing is to keep *IssueTime* recent.

This is accomplished by adding another bit to each reorder buffer (ROB) entry to track the MSB of its DispatchTime (i.e., the time at which the micro-op was dispatched to the ROB). So, each ROB entry tracks if it was dispatched during the even or the odd quantum. If the DispatchTime is kept 'recent', the *IssueTime* of a load operation will also be recent, since issue is after dispatch. The DispatchTime is kept recent by monitoring the entry at the head of the ROB and the timestamp counter (TC). If the TC rolls over from the odd to the even quantum with the head of the ROB pointing to an entry dispatched during the even quantum, then that entry's timestamp is considered 'old'. A speculation violation is triggered and instructions are replayed starting with the one at the head of the ROB. Likewise, if the TC rolls over from the even to the odd quantum with the head pointing to an odd quantum entry, a consistency violation is triggered. Through experimental observations, the timestamp width (TW) is set to 16 bits. This keeps the storage overhead manageable while creating almost no false positives. With $TW = 16$, each entry in the ROB has $2^{TW-1} = 32768$ cycles to commit before a consistency violation is triggered due to rollover.

E. Mixing Remote Accesses and Private Caching

The previous sections described the implementation of TSO on a pure remote access scheme. The locality-aware protocol chooses either remote access at the shared L2 cache or private caching at the L1 cache based on the spatio-temporal locality of data. Hence, the timestamp-based consistency validation scheme should be adapted to such a protocol.

1) L1 Cache History Queues (L1LHQ/L1SHQ): Such an adaptation requires information about loads and stores made to the private L1-D cache to be maintained for future reference in order to perform consistency validation. This information

needs to be captured because private L1-D cache loads/stores can execute out-of-order and interact with either remote or private cache accesses such that the TSO memory consistency model is violated. Similar to the history queues at the L2 cache, the L1 Load History Queue (L1LHQ) and the L1 Store History Queue (L1SHQ) are added at the L1-D cache (shown in Figure 1) and capture the load and store history respectively. The history retention period (HRP) dictates how long the history is retained for. The management of the L1LHQ/L1SHQ (i.e., adding/pruning/searching) is carried out in the exact same manner as the L2 history queues.

With history queues at multiple levels of the cache hierarchy, it is important to keep them synchronized. An invalidation, downgrade, or eviction request at the L1-D cache causes the last load/store timestamps (if found) to be sent back along with the acknowledgement so that they can be preserved at the shared L2 cache history queues until the history retention period (HRP) expires. From the L2LHQ/L2SHQ, these load/store timestamps are passed onto cores that remotely access or privately cache data. A cache line fetch from the shared L2 cache into the L1-D cache copies the load/store history for the address into the L1LHQ/L1SHQ as well. This enables the detection of consistency violations using the same timestamp-based validation technique described earlier.

2) Exclusive Store Prefetch: Since exclusive store prefetch requests can be used to improve the performance of stores that are cached at the L1-D cache, they must be leveraged by the locality-aware protocol as well. In fact, these prefetch requests can be leveraged by remote stores to prefetch cache lines from off-chip DRAM into the L2 cache. This can be accomplished only if both private and remote stores are executed in two phases.

The first phase (exclusive store prefetch) is executed in parallel and potentially out-of program order as soon as the store address is ready. If the cache line is already present at the L2 cache, then the first phase for remote stores is effectively a NOP but must be executed nevertheless since the information about whether a store is handled remotely or cached privately is only present at the directory (that is co-located with the shared L2 cache). The second phase (actual store) is executed in order, i.e., the store is issued only after all previous stores have completed (to ensure TSO ordering) and the first phase of the current store has been acknowledged. The second phase for stores to the private L1-D cache complete quickly ($\sim$1-2 cycles), while remote stores have to execute a round-trip network traversal to the remote L2 cache to be completed.

F. Parallelizing Non-Conflicting Accesses

An alternate/complementary method to exploit memory level parallelism while maintaining TSO is to recognize the fact that only conflicting accesses to shared read-write data can cause memory consistency violations. Concurrent reads to shared read-only data and accesses to private data cannot lead to violations [14], [12]. Such memory accesses can be both issued and completed out-of-order. Only accesses to shared read-write data must be ordered according to the TSO model. In order to accomplish the required classification of data into private, shared read-only and shared read-write, our baseline machine [12] extends the page-level classifier by augmenting existing TLB and page table structures (similar to [14]).

The TSO ordering of shared read-write data could be implemented in two ways: (1) executing memory operations in the strict order mandated by TSO as described in Section III-A, and (2) employing the timestamp-based speculative execution scheme discussed in the previous subsections. Note that using the timestamp check only for shared read-write data implies that the history queue modifications and search operations can be avoided for private and shared read-only data. This reduces the energy overhead of the history queues. We will evaluate both these approaches in this paper.

G. Overheads

The storage, latency and energy overheads of the timestamp-based technique are as follows:

L1 History Queues: The L1SHQ (L1 store history queue) is sized based on the expected throughput of store requests to the private L1-D cache and the History Retention Period (HRP). In Section VI-C, HRP is empirically found to be $512ns$ using a sensitivity study. A memory access is expected every 3 instructions, and a store is expected every 3 memory accesses, so for a single issue processor with a 1 GHz clock, a store is expected every $9ns$. Each L1SHQ entry contains the store timestamp and the physical cache line address (42 bits). The width of each timestamp is 16 bits (as discussed above). Hence, the size of the L1SHQ $= \frac{512\times(16+42)}{9}$bits $= 0.4$KB. The throughput of loads is approximately twice that of stores, hence the size of the L1LHQ (L1 load history queue) is 0.8KB.

Since the L1LHQ and L1SHQ are much smaller than the L1-D cache, they can be accessed in parallel with the cache tags, and so do not add any extra latency. The energy expended when accessing these structures is modeled in our evaluation.

L2 History Queues: The L2SHQ is sized based on the expected throughput of remote store requests to the L2 cache and invalidations/write-backs from the L1-D cache. The throughput of remote requests is much less than that of private L1-D cache requests, but can be susceptible to higher contention if many remote requests are destined for the same L2 cache slice. To be conservative, the expected throughput is set to one store every 18 processor cycles (this is $4\times$ the average expected throughput from experiments). The same calculation (listed above) is repeated to obtain a size of 0.2KB. The L2LHQ has twice the expected throughput as the L2SHQ so its size is 0.4KB.

Since the L2LHQ and L2SHQ are much smaller than the L2 cache, they can be accessed in parallel with the cache tags, and do not add any extra latency. The energy expended when accessing these structures is modeled in our evaluation.

Load/Store Queues and Reorder Buffer: Each load queue is augmented with 2 timestamps and each store queue entry with 1 timestamp respectively (as shown in Figures 3 and 4). With 64 load queue entries, the overhead is $64 \times 2 \times 16$ bits $=$ 256 bytes. With 48 store queue entries, the overhead is 48×16 bits $= 96$ bytes. A single bit added to the ROB for timestamp overflow detection has only a negligible overhead.

Network Traffic: Whenever an entry is found in the L1LHQ/L1SHQ on an invalidation/write-back request or an

entry is found in the L2LHQ/L2SHQ during a remote access or cache line fetch from the L2 cache, the corresponding timestamp is added to the acknowledgement message. Since each timestamp width is 16 bits and the network flit size is 64 bits (see Table I), even if both the load and store timestamps need to be accommodated, only 1 extra flit needs to be added to the acknowledgement.

Counting all the above changes, the total storage overhead is $\sim 2.2KB$ per core. We consider the latency and energy overheads associated with the timestamp scheme in our evaluation.

H. Forward Progress and Starvation Freedom Guarantees

Forward progress for each core is guaranteed by the timestamp-based consistency validation protocol. To understand why, consider the two reasons why load speculation could fail: (1) Consistency Check Violation, and (2) Timestamp Rollover.

If speculation fails due to the consistency check violation, then re-executing the load is guaranteed to allow it to complete. This is because the *IssueTime* of the load (when executed for the second time) will always be greater than the time at which the consistency check was made, i.e., the commit time of the load, which in turn is greater than *OrderingTime*. This is because *OrderingTime* is simply the maximum of load and store timestamps observed by previous memory operations, and the time at which the load is committed is trivially greater than this.

If speculation fails due to timestamp rollover, then re-executing the load/stores is guaranteed to succeed unless the timestamp rolls over again. This condition is prevented by adding a special check to make sure the operation where re-execution starts always commits (regardless of rollover) since it cannot conflict with any previous operation. Since forward progress is guaranteed for all the cores in the system, this technique of ensuring TSO ordering is *starvation-free*.

IV. Discussion

A. Other Memory Models

Section III discussed how to implement the TSO memory ordering using the proposed timestamp-based memory consistency verification scheme. The TSO model is the most popular, being employed by x86 and SPARC processors. Other memory models of interest are Sequential Consistency (SC), Partial Store Order (PSO), and the IBM Power/ARM models. We provide an overview of how they can be implemented with the timestamp-based scheme.

Sequential Consistency (SC) can be implemented by associating an implicit fence after every store operation. Hence, in the CompleteStore function in Section III-B, each store directly updates *OrderingTime* using its *LastAccessTime*. This ensures that the Store $\rightarrow$Load program order is maintained in the global memory order.

Partial Store Order (PSO) relaxes the Store $\rightarrow$Store ordering and only enforces it when a fence is present. This enables all stores, both private and remote, to be issued in parallel and potentially completed out-of-order. On a fence that enforces Store $\rightarrow$Store ordering, stores after a fence can be issued only after the stores before it complete.

IBM Power is a more relaxed model that enforces minimal ordering between memory operations in the absence of fences. Here, we discuss how its two main fences, `lwsync` and `hwsync` are implemented. `lwsync` enforces TSO ordering and can be implemented by maintaining a *LoadOrderingTime* field that keeps track of the maximum *LastStoreTime* observed so far. On a fence, the *LoadOrderingTime* is copied to the *OrderingTime* field and the timestamp checks outlined earlier are run. `hwsync` enforces SC ordering. This can be implemented by taking the maximum of the *LoadOrderingTime* and *StoreOrderingTime* and updating the *OrderingTime* field with this value. The ARM memory model is similar to the IBM Power model and hence can be implemented in a similar way.

B. Multiple Clock Domains

The assumption in Section III was that there is a single clock domain in the system. However, current multicore processors are gravitating towards multiple voltage and clock domains with independent dynamic frequency scaling (DVFS). In such processors, keeping timestamps synchronous is challenging. We assume that a common global clock would be available in this system, for example, to ensure a capability for deterministic debug of the processor. The timestamps can be managed using this global clock. In summary, a thorough evaluation of supporting multiple clock domains is a challenging problem. We defer an evaluation of this aspect to future work.

V. Evaluation Methodology

We evaluate a 64-core processor. The default architectural parameters used for evaluation are shown in Table I.

A. Performance Models

All experiments are performed using the out-of-order core, cache hierarchy, coherence protocol, memory system and on-chip interconnection network models implemented within the Graphite [15] multicore simulator. All the mechanisms and protocol overheads discussed are modeled. Graphite uses a barrier mechanism to synchronize the execution of cores. We use a barrier interval of $100ns$, i.e., the simulations of all the cores are synchronized after every interval of $100ns$. Within each interval, the cores could get ahead of each other but by $100ns$ at most. In order to prevent this from affecting results, a history of all load and store access times to each cache level is maintained for at least $100ns$ (and at most for the history retention period), and the load/store times within this history are used to detect consistency violations.

Each out-of-order core is modeled with an issue width of one instruction per cycle. However, the core implements a 128-entry reorder buffer to enable out-of-order and speculative scheduling of instructions. A 64-entry load queue and a 48-entry store queue also ensures each core's ability to exploit memory level parallelism.

B. Energy Models

We evaluate just dynamic energy. For energy evaluations of on-chip electrical network routers and links, we use the DSENT [20] tool. Energy estimates for the L1-I, L1-D and L2 (with integrated directory) caches are obtained using McPAT [21]. The evaluation is performed at the $11nm$ technology node to account for future technology trends.

Architectural Parameter	Value
Number of Cores	64 @ 1 GHz
Physical Address Length	48 bits
Core	
Type	Out-of-order, Single-issue
Reorder Buffer Size	128
Load Queue Size	64
Store Queue Size	48
Speculation Violation	Timestamp-based
Memory Subsystem	
L1-I Cache per core	16 KB, 4-way Assoc., 1 cycle
L1-D Cache per core	32 KB, 4-way Assoc., 1 cycle
L2 Cache per core	256 KB, 8-way Assoc., 6 cycles Inclusive, R-NUCA
Cache Line Size	64 bytes
Directory Protocol	Invalidation-based MESI ACKwise$_4$ [13]
Num. of Memory Controllers	8
DRAM Bandwidth	5 GBps per Controller
DRAM Latency	75 ns
Electrical 2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Contention Model	Only link contention (Infinite input buffers)
Flit Width	64 bits
Header (Src, Dest, Addr, MsgType)	1 flit
Word Length	1 flit (64 bits)
Cache Line Length	8 flits (512 bits)
Locality-Aware Coherence Protocol [10]	
Private Caching Threshold	*PCT* = 4

TABLE I
ARCHITECTURAL PARAMETERS FOR EVALUATION.

The energy consumption during the INSERT and SEARCH operations of each history queue is conservatively assumed to be the amount of energy it takes for an L1-D cache tag read. This is justified since the size of the L1-D cache tag array is $2.6KB$. The tag array contains 512 tags, each 36 bits wide (subtracting out the index and offset bits from the physical address). On the other hand, the size of each history queue is $\leq 0.8KB$.

C. Application Benchmarks

We simulate six SPLASH-2 [16] benchmarks, six PARSEC [17] benchmarks, one Parallel-MI-Bench [18], and ten CRONO graph analytics benchmarks [19]. Each multithreaded benchmark is run to completion using the input sets from Table II.

Application	Problem Size
SPLASH-2 [16]	
RADIX	4M Integers, radix 1024
LU	512 × 512 matrix, 16 × 16 blocks
BARNES	64K particles
OCEAN	514 × 514 ocean
WATER	512 molecules
VOLREND	head
PARSEC [17]	
BLACKSCHOLES	64K options
SWAPTIONS	64 samples, 40,000 times
DEDUP	31 MB data
BODYTRACK	2 frames, 2000 particles
FACESIM	1 frame, 372,126 tetrahedrons
CANNEAL	200,000 elements
Parallel MI Bench [18]	
PATRICIA	5000 IP address queries
CRONO [19]	
BFS, DFS, PAGERANK, SSSP DIJKSTRA, TRIANGLE COUNTING, CONNECTED COMPONENTS, COMMUNITY DETECTION	Graph with 2^{18} nodes, 16 edges/node
ALL PAIRS SHORTEST PATH, BETWEENNESS CENTRALITY	Graph with 2^{12} nodes, 16 edges/node
TSP	16 cities

TABLE II
PROBLEM SIZES FOR THE PARALLEL BENCHMARKS.

VI. RESULTS

A. Comparison of Schemes

In this section, we perform an exhaustive comparison between the various schemes introduced in this paper to implement locality-aware coherence on an out-of-order processor while maintaining the TSO memory model. The comparison is performed against the Reactive-NUCA protocol. All implementations of the locality-aware protocol use a *PCT* value of 4. Section VI-B describes the rationale behind this choice.

1) Reactive-NUCA (RNUCA): This is the baseline scheme that implements the data placement and migration techniques of R-NUCA (basically, the locality-aware protocol with a *PCT* of 1).
2) Simple TSO Implementation (SER): The simplest implementation of TSO on the locality-aware protocol that serializes memory accesses naively according to TSO ordering (cf. Section III-A).
3) Parallel Non-Conflicting Accesses (NC): This scheme classifies data as shared/private and read-only/read-write at page-granularity and only applies serialization to shared read-write data (cf. Section III-F).
4) Timestamp-based Consistency Validation (TS): Executes loads speculatively using timestamp-based validation (cf. Section III) for shared read-write data. Shared read-only and private data are handled as in the NC scheme.
5) Timestamp-based + Stall Fence (TS-STF): Same as TS but the micro-op dispatch is stalled till a fence completes to ensure Fence→Load ordering. The load history queues (L1LHQ and L2LHQ) are not required for detecting violations here. It has lower hardware overhead than TS but potentially lower performance due to stalling on fence operations.
6) No Speculation Violations (IDEAL): Same as TS but speculation violations are ignored. It provides the upper limit on performance and energy consumption. The L1 and L2 history queues are not required since no speculation failure checks are made.

The completion time and energy consumption of the above schemes are plotted in Figures 6 and 7 respectively. The distribution of completion time and energy between the caches and network varies across benchmarks and is primarily dependent on the private L1 cache miss rate. For this purpose, the L1

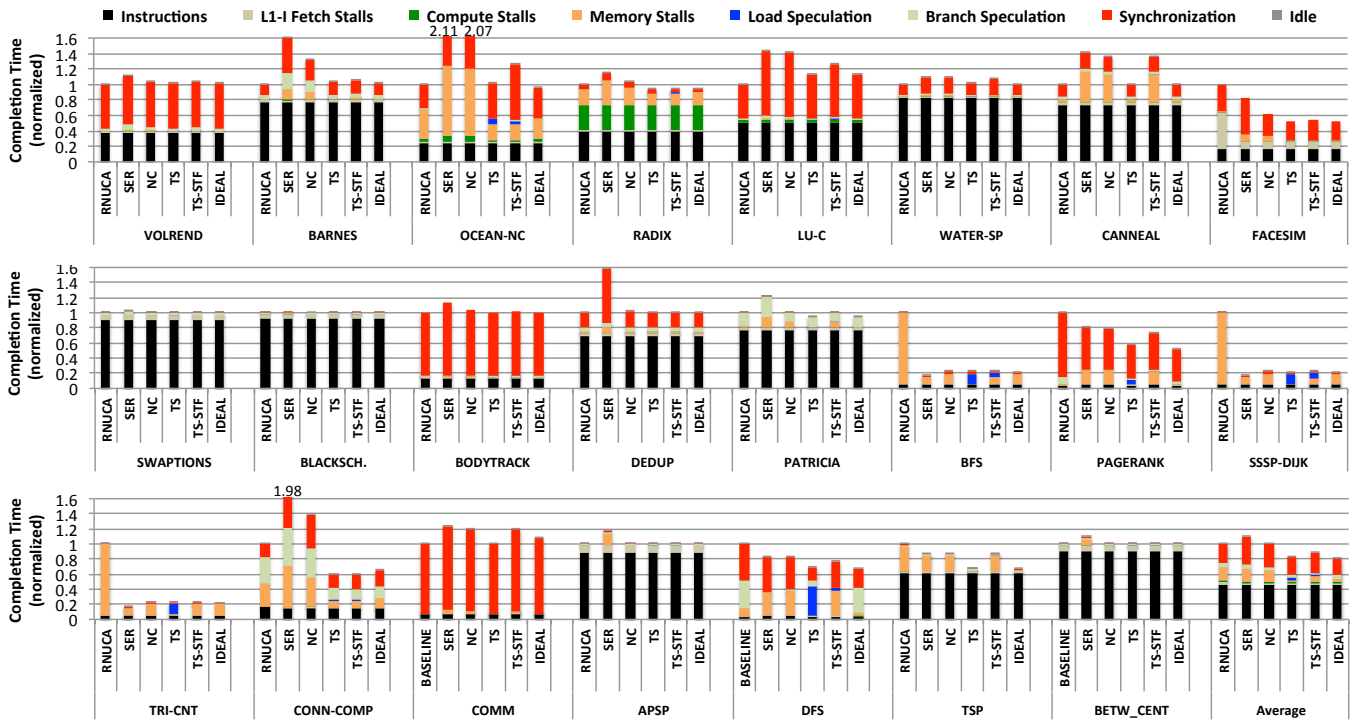

Fig. 6. Completion Time breakdown for the schemes evaluated. Results are normalized to that of Reactive-NUCA (RNUCA). Note that *Average* and not *Geometric-Mean* is plotted here.

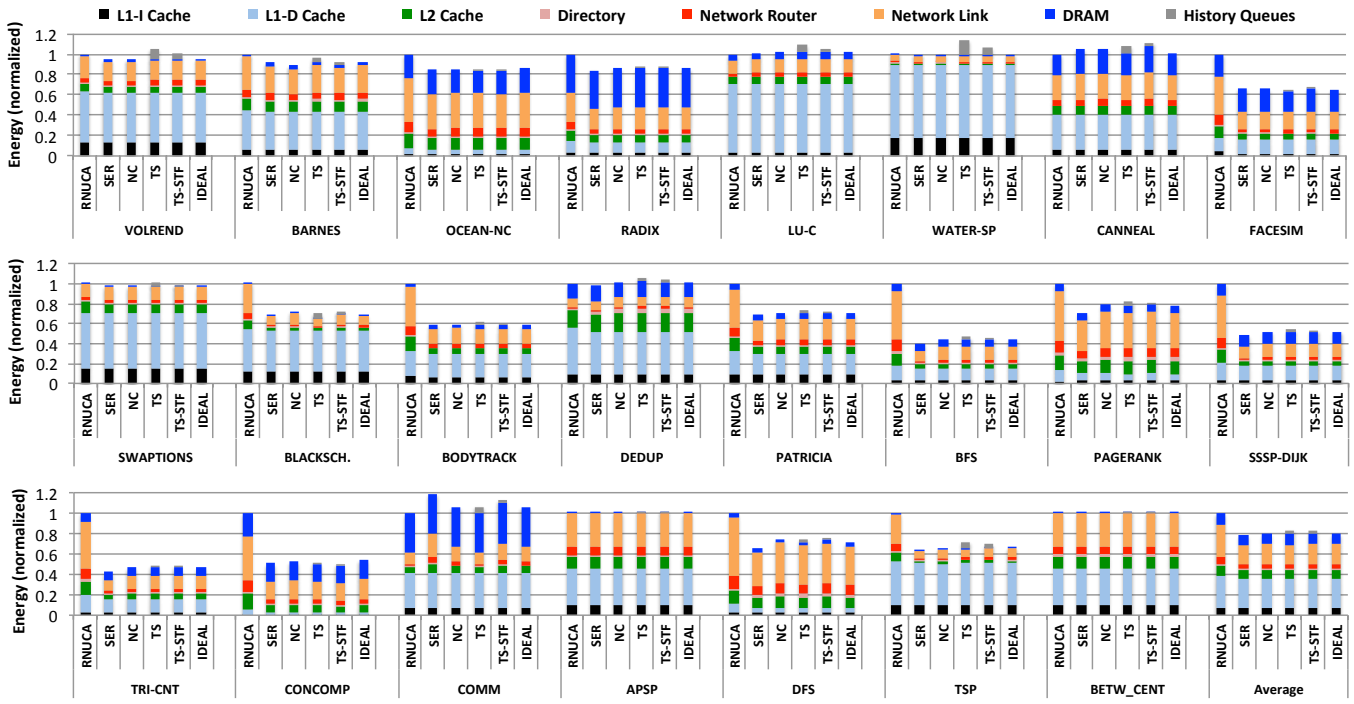

Fig. 7. Energy breakdown for the schemes evaluated. Results are normalized to that of RNUCA. Note that *Average* and not *Geometric-Mean* is plotted here.

cache miss rate is plotted along with miss type breakdown in Figure 8.

Completion Time: The parallel completion time is broken down into the following categories:

1) *Instructions:* Time spent retiring instructions.
2) *L1-I Fetch Stalls:* Stall time due to instruction cache misses.
3) *Compute Stalls:* Stall time due to waiting for functional unit (ALU, FPU, Multiplier, etc.) results.
4) *Memory Stalls:* Stall time due to load/store queue capacity limits, fences and waiting for load completion.
5) *Load Speculation:* Stall time due to memory consistency

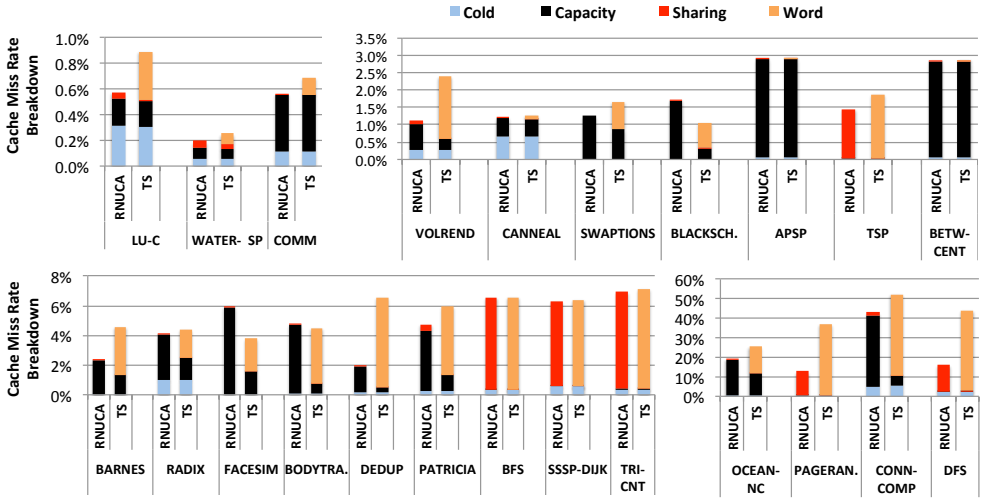

Fig. 8. Private L1 Cache Miss Rate and Miss Type Breakdown for the TS and Reactive-NUCA (RNUCA) schemes. Note that in this graph, the miss rate increases from left to right as well as from top to bottom.

violations caused by speculative loads.

6) *Branch Speculation:* Stall time due to mis-predicted branch instructions.
7) *Synchronization:* Stall time due to waiting on locks, barriers and condition variables.
8) *Idle:* Initial time spent waiting for parallel worker threads to be spawned.

Benchmarks with low private cache miss rate (LU-C, WATER-SP, COMM and CANNEAL) do not gain from the locality-aware protocol since only a small number of sharing misses are converted to word misses. However, these benchmarks contain a significant degree of synchronization. This causes the memory stalls in one thread to increase the synchronization penalty of threads waiting on it, thereby creating a massive slowdown for the SER scheme. The performance problems observed by the SER scheme are shared by the NC scheme as well. The NC scheme can only efficiently handle accesses to private and shared read-only data. Since these benchmarks contain a majority of accesses to shared read-write data, the NC scheme performs poorly. The TS scheme allows speculative cores to hide the latency of all L1 cache misses and hence the performance of these benchmarks stay competitive with respect to the RNUCA baseline. The only exception is LU-C benchmark that observes a significant slowdown due to store queue capacity stalls created by serializing remote stores. The TS-STF scheme stalls the issue of load operations on a fence till all previous stores have committed. It performs poorly in these benchmarks due to significant numbers of fences. Note that all fences seen in the evaluated benchmarks are implicit fences introduced by atomic operations in x86, e.g., test-and- set, compare-and-swap, etc. There were almost no explicit MFENCE instructions observed.

Several benchmarks, VOLREND, SWAPTIONS, BARNES and DEDUP convert private L1 capacity misses into word misses. Each remote access generates lower network traffic compared to a capacity miss. However, these benchmarks convert a capacity miss into *multiple* word misses since the cache lines are reused a few times ($<$ PCT of 4) before eviction. The memory stalls due to accesses to remote lines create performance slowdowns with the SER scheme. These slowdowns are again worse when memory stalls fall within critical sections that synchronize threads, as seen clearly for BARNES and DEDUP. The NC scheme performs on par with the TS scheme for these benchmarks except in BARNES. This is because the majority of L1 cache capacity misses in BARNES are for read-write shared data while they are for private and read-only shared data in the other 3 benchmarks. Like TS, the NC scheme exploits MLP for access to non-conflicting data and hence performs almost on par with the RNUCA scheme.

Benchmarks with a high L1 cache miss rate (OCEAN-NC and CONN-COMP) do not perform well with the SER scheme. This is because the cache misses cause the load and store queues to fill up, thereby stalling the pipeline. This can be understood by observing the fraction of memory stalls in the completion time of these benchmarks (with the SER scheme). In addition, these benchmarks contain a significant degree of synchronization. The TS scheme only spends a small amount of time stalling due to load speculation violations. This stalling due to speculation violation just replaces the already occurring stalls due to the limited size of the reorder buffer. Moreover, since remote accesses are much cheaper than sharing and capacity misses, they help reduce memory stalls in these benchmarks. This results in an overall performance gain that is clearly observed for the CONN-COMP benchmark. Similar trends are seen for the RADIX, PATRICIA and BODYTRACK benchmarks.

The BLACKSCHOLES and FACESIM benchmarks convert capacity misses into word misses. This improves cache utilization (i.e., reduces cache pollution) and thereby lowers capacity misses for other cache lines. This results in a lower L1 cache miss rate compared to the RNUCA baseline. Moreover, a significant proportion of these misses are much cheaper word misses. As a consequence of these two factors, all schemes that use the locality-aware protocol improve performance, as clearly evident in the FACESIM benchmark. The TS scheme delivers the highest performance gain since it fully exploits the speculative execution of the cores under the TSO memory model.

Benchmarks with significant L1 sharing miss rate (TSP, BFS, DFS, SSSP-DIJK, TRI-CNT, and PAGERANK) perform significantly well for all schemes. From a performance standpoint, sharing misses are expensive because they incur additional network traffic generated by invalidations and synchronous write-backs. In these benchmarks, even if cache miss rate increases with remote accesses, the miss penalty is lower because a word miss is much cheaper than a sharing miss. This results in a significant reduction in the memory stalls for these benchmarks. Reducing the memory stalls may decrease synchronization time as well if the responsible memory accesses lie within the critical section. This can be observed in the DFS and PAGERANK benchmarks. On the downside, the TS scheme now spends time stalling due to load speculation violations. However, this stalling accounts for much less time than the memory stalls in the RNUCA baseline and hence, these benchmarks benefit greatly in performance.

Overall, the TS scheme performs well on our benchmarks and matches the performance of the IDEAL scheme. The TS, TS-STF and NC schemes improve performance over the RNUCA baseline by a geometric mean of 26%, 20% and 13% (average of 18%, 11% and 1%) respectively. The SER scheme reduces performance by an average of 9%.

Energy: All the locality-aware coherence protocol implementations except for RNUCA are found to significantly reduce L2 cache and network energy due to the following three factors:

1) Fetching an entire line on a cache miss is replaced by multiple cheaper word accesses to the shared L2 cache.
2) Reducing the number of private sharers decreases the number of invalidations (and acknowledgments) required to keep all cached copies of a line coherent. Synchronous write-back requests that are needed to fetch the most recent copy of a line are reduced as well.
3) Since the caching of low-locality data is eliminated, the L1 cache space is more effectively used for high locality data, thereby decreasing the amount of asynchronous evictions (that lead to capacity misses) for such data.

Among the locality-aware coherence protocol implementations, SER, NC, and IDEAL exhibit the best dynamic energy consumption. Dynamic energy consumption increases when TS-STF is used and increases even further when TS is used. This is because both these implementations modify and access the L1 and L2 cache history queues. The TS-STF scheme only requires the store history queues since it stalls on a fence while the TS scheme requires both load and store history queues to perform consistency checks, thereby creating a larger energy overhead. Note that page-classification is used in both the TS and TS-STF schemes to ensure that history queue modification and access is only done for shared read-write data since accesses to private and shared read-only data cannot cause consistency violations. Overall, the TS, TS-STF, SER and NC schemes reduce energy by a geometric mean of 20%, 20.5%, 24% and 22% (average of 16.5%,17.5%, 20% and 20%) respectively over the RNUCA baseline.

B. Sensitivity to Private Caching Threshold (PCT)

In this section, we study the impact of the Private Caching Threshold (*PCT*) parameter on the overall system performance. *PCT* controls the percentage of remote and private

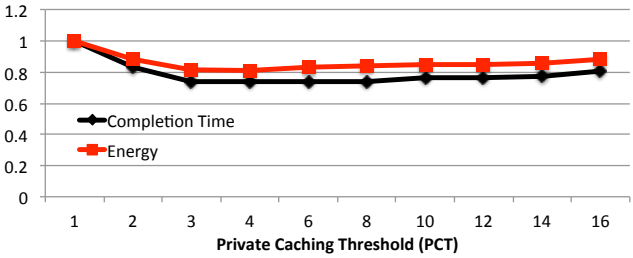

Fig. 9. Variation of the *Geometric-Means* of Completion Time and Energy with Private Caching Threshold (PCT). Results are normalized to a *PCT* value of 1 (i.e., the Reactive-NUCA protocol).

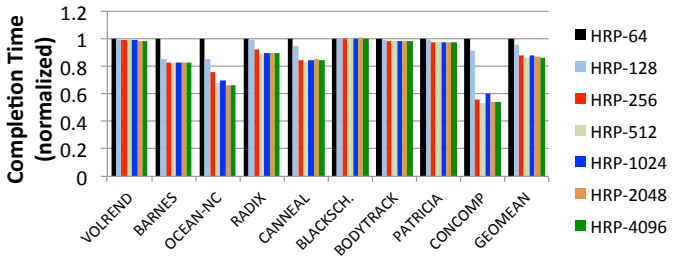

Fig. 10. Completion Time sensitivity to History Retention Period (HRP) as HRP varies from 64 to 4096.

cache accesses in the locality-aware protocol. A higher *PCT* increases the percentage of remote accesses while a lower *PCT* increases the percentage of cache line fetches into the private cache. Finding the optimal value of *PCT* is of paramount importance to system performance. We plot the geometric means of the *Completion Time* and *Energy* for our benchmarks as a function of *PCT* in Figure 9. We observe a gradual decrease in completion time till a *PCT* of 3, constant completion time till a *PCT* of 8 and then a gradual increase afterward. Energy consumption reduces steadily till a *PCT* of 4, reaches a global minimum at 4 and then increases steadily afterward.

The completion time shows an initial gradual reduction due to lower network contention. The gradual increase afterward is due to increased stall time from remote loads. Overall, at *PCT* of 4, the locality-aware protocol obtains a 26% completion time reduction and a 20% energy reduction when compared to the Reactive-NUCA baseline.

C. Sensitivity to History Retention Period (HRP)

In this section, the impact of the History Retention Period (HRP) on system performance is studied. Figure 10 plots the completion time as a function of HRP. A small value of HRP reduces the size requirement of the load/store history queues at the L1 and L2 caches (L1LHQ, L1SHQ, L2LHQ and L2SHQ) as described in Section III-G. A small HRP also reduces network traffic since timestamps are less likely to be found in the history queues, and thereby less likely to be communicated in a network message. However, a small HRP also discards history information faster, requiring the mechanism to make a conservative assumption regarding the time the last loads/stores were made (cf. Section III-C). This increases the chances of the speculation check failing, thereby increasing completion time.

From Figure 10, we observe that an HRP of 64 performs

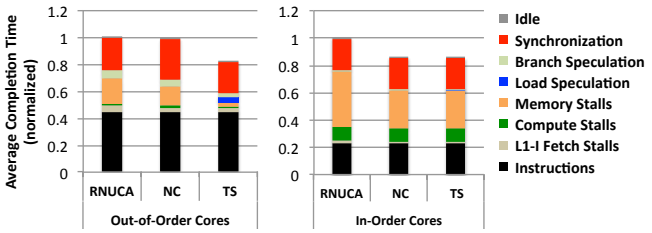

Fig. 11. Average Completion Time breakdown for the RNUCA, NC and TS schemes. Results are normalized to RNUCA for each core type.

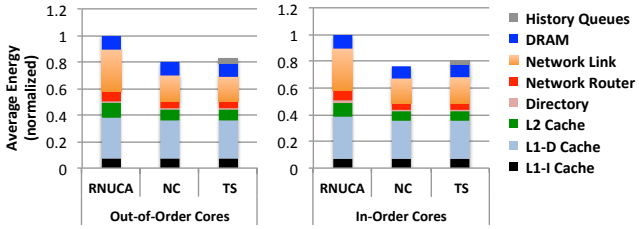

Fig. 12. Average Energy breakdown for the RNUCA, NC and TS schemes. Results are normalized to RNUCA for each core type.

considerably worse when compared to other data points. The high completion time is due to the instruction replays incurred due to speculation violation. HRP values of 128 and 256 reduce speculation violations, and thereby improve performance considerably. However, we opted an HRP of 512 since its performance is within $\sim$1% of a large HRP of 4096.

D. Sensitivity to Out-of-Order vs In-Order Core Types

In this section, the impact of the type of core used in a multicore processor is studied. Figures 11 and 12 show the average completion time and energy results for out-of-order and in-order core based multicore processors. It is immediately apparent that the percentage of time spent in memory and compute stalls is much lower in out-of-order processors due to their dynamic scheduling. However, the energy profiles are quite similar (since only dynamic energy is modeled). We observe that an out-of-order system requires the TS scheme to improve performance with the locality-aware protocol. The NC scheme incurs load/store serialization stalls since speculative execution of loads to shared read-write data is not supported and cannot be used to improve performance.

On the other hand, an in-order core issues and commits all instructions in program order. Long latency operations such as private cache misses cannot be hidden and create pipeline stalls when a dependent instruction is encountered. Hence, the capability of the in-order core to exploit MLP is limited. Speculative execution of load operations is not expected to be much beneficial and this is confirmed by the performance results which indicate that there is no advantage of using the TS scheme over the much simpler NC scheme. The energy consumption is worse for the TS scheme since it needs to access/update the load/store history queues. However, both the NC and TS schemes reap the benefits of the locality-aware protocol. Overall, the in-order system with NC scheme improves completion time by 23% and energy by 27% over the RNUCA baseline. (Note: the improvements reported here use geometric mean versus average in Figures 11 and 12.)

VII. Related Work

A. Timestamps

Timestamps have been used previously to implement cache coherence [22], [23], [24], and coherence verification [25]. The idea of using sequence numbers or timestamps to create and track ordering between memory operations has also been explored [26], [27]. Our paper is unique in the following ways: (1) it provides a detailed design of an *implementable* timestamp scheme to detect consistency violations in modern multicores; (2) it applies the timestamp scheme on top of a state-of-the-art directory based locality-aware cache coherence protocol; (3) it implements the timestamp scheme in a functionally correct multicore simulator that utilizes the popular x86 ISA and TSO memory model; (4) it shows that the timestamp scheme can be implemented with a lightweight hardware overhead of $\sim 2.2KB$ per core.

B. Memory Models

Speculatively relaxing memory order was proposed with load speculation in MIPS R10K [28]. Full speculation for SC execution has been studied as well [29], [14]. InvisiFence [30], Store-Wait-Free [31] and BulkSC [32] are proposals that attempt to accelerate existing memory models by reducing both buffer capacity and ordering related stalls. In this paper, we have evaluated a novel timestamp-based scheme to detect memory consistency violation for the popular TSO memory model.

C. Remote (Cache) Access

Remote access has been used as the sole mechanism to support shared memory in multicores [33], [34], [11]. Recently, research proposals that utilize remote access as an auxiliary mechanism [35], [10] have demonstrated improvement in performance and energy consumption for large-scale multicores. In this paper, we observe that complex cores that support popular memory models (e.g., x86 TSO, ARM, SC) need novel mechanisms to benefit from these adaptive protocols.

D. Techniques to Improve Performance and Energy

Data access and network bottlenecks in multicores can be alleviated using intelligent cache hierarchy management. Better last-level cache (LLC) partitioning [36], [37] and replacement schemes [38], [39] have been proposed to reduce memory pressure. Better cache replication [12], [40], [41], placement [12], [37] and allocation [10] schemes have been proposed to exploit application data locality and reduce network traffic. Our proposed extension for locality-aware cache coherence ensures that it can be implemented in multicore processors with popular memory models alongside any of the above schemes.

VIII. Conclusion

In this paper we propose a timestamp-based memory consistency verification scheme that enables invalidation-free data access protocols to execute efficiently using speculation. The scheme continuously detects whether any ordering violations have occurred and rolls back the pipeline state (if needed). We implement our scheme for a state-of-the-art locality-aware cache coherence protocol that uses remote access as an auxiliary mechanism for efficient data access. Our evaluation using

a 64-core multicore with out-of-order speculative cores shows that our proposed technique improves completion time by 26% and energy by 20% over the RNUCA cache management scheme.

IX. Acknowledgment

This research was partially supported by the National Science Foundation under Grant No. CCF-1452327. We would also like to thank the anonymous reviewers for their constructive feedback.

References

[1] "Intel launches low-power, high-performance silvermont microarchitecture," Intel.com, 2013.

[2] P. Greenhalgh, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," ARM White Paper, 2011.

[3] J. Reinders, "Knights Corner: Your Path to Knights Landing," Intel Developer Zone, October 2014.

[4] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[5] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams, "An axiomatic memory model for power multiprocessors," in *Computer Aided Verification (CAV)*, 2012.

[6] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, "Don't sit on the fence: A static analysis approach to automatic fence insertion," in *Computer Aided Verification (CAV)*, 2014.

[7] 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, Jul. 2010.

[8] "The sparc architecture manual, v. 8. sparc international, inc," http://www.sparc.org/standards/V8.pdf, 1992.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *International Conference on Parallel Processing (ICPP)*, 1991.

[10] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[11] C. Fensch and M. Cintra, "An os-based alternative to full hardware coherence on tiled cmps," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[12] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," in *International Symposium on Computer Architecture (ISCA)*, 2009.

[13] G. Kurian, J. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. Kimerling, and A. Agarwal, "ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[14] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end Sequential Consistency," in *International Symposium on Computer Architecture (ISCA)*, 2012.

[15] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

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

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

[18] S. Iqbal, Y. Liang, and H. Grahn, "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems," *Computer Architecture Letters*, vol. 9, no. 2, pp. 45 –48, feb. 2010.

[19] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores," in *IEEE International Symposium on Workload Characterization, (IISWC)*, 2015.

[20] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," in *International Symposium on Networks-on-Chip (NOCS)*, 2012.

[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *International Symposium on Microarchitecture (MICRO)*, 2009.

[22] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[23] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[24] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood, "Timestamp Snooping: An Approach for Extending SMPs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[25] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill, "Lamport Clocks: Verifying a Directory Cache-coherence Protocol," in *Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998.

[26] A. Meixner and D. Sorin, "Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 1, pp. 18–31, Jan 2009.

[27] T. Arons, "Using timestamping and history variables to verify sequential consistency," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2001, vol. 2102, pp. 423–435.

[28] K. Yeager, "The Mips R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–41, Apr 1996.

[29] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC?" in *International Symposium on Computer Architecture (ISCA)*, 1999.

[30] C. Blundell, M. Martin, and T. F. Wenisch, "InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 2009.

[31] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for Store-wait-free Multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 2007.

[32] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *International Symposium on Computer Architecture (ISCA)*, 2007.

[33] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[34] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming: A memory model for embedded multicore," in *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2010.

[35] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim, "Location-aware Cache Management for Many-core Processors with Deep Cache Hierarchy," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[36] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *International Symposium on Microarchitecture (MICRO)*, 2006.

[37] N. Beckmann and D. Sanchez, "Jigsaw: Scalable Software-defined Caches," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[38] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *International Symposium on Computer Architecture (ISCA)*, 2007.

[39] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[40] M. Zhang and K. Asanović, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 2005.

[41] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive Selective Replication for CMP Caches," in *International Symposium on Microarchitecture (MICRO)*, 2006.