Reaping the Benefit of Temporal Silence to Improve Communication Performance

Kevin M. Lepak* and Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin
Madison, WI 53706
{lepak,mikko}@ece.wisc.edu
*Now with Advanced Micro Devices

Abstract

Communication misses--those serviced by dirty data in remote caches--are a pressing performance limiter in shared-memory multiprocessors. Recent research has indicated that temporally silent stores can be exploited to substantially reduce such misses, either with coherence protocol enhancements (MESTI); by employing speculation to create atomic silent store-pairs that achieve speculative lock elision (SLE); or by employing load value prediction (LVP). We evaluate all three approaches utilizing full-system, execution-driven simulation, with scientific and commercial workloads, to measure performance. Our studies indicate that accurate detection of elision idioms for SLE is vitally important for delivering robust performance and appears difficult for existing commercial codes. Furthermore, common datapath issues in out-of-order cores cause barriers to speculation and therefore may cause SLE failures unless SLE-specific speculation mechanisms are added to the microarchitecture. We also propose novel prediction and silence detection mechanisms that enable the MESTI protocol to deliver robust performance for all workloads. Finally, we conduct a detailed execution-driven performance evaluation of load value prediction (LVP), another simple method for capturing the benefit of temporally silent stores. We show that while theoretically LVP can capture the greatest fraction of communication misses among all approaches, it is usually not the most effective at delivering performance. This occurs because attempting to hide latency by speculating at the consumer, i.e. predicting load values, is fundamentally less effective than eliminating the latency at the source, by removing the invalidation effect of stores. Applying each method, we observe performance changes in application benchmarks ranging from 1% to 14% for an enhanced version of MESTI, -1.0% to 9% for LVP, -3% to 9% for enhanced SLE, and 2% to 21% for combined techniques.

1 INTRODUCTION

All modern shared-memory multiprocessors rely on invalidation protocols to maintain cache coherence; in such a protocol a local write causes all remote copies of the block to be invalidated [12]. Subsequent remote reads result in cache-to-cache transfers of dirty data (known as communication misses or dirty misses); most researchers and practitioners agree that reducing the frequency and latency of such misses is vitally important to guarantee high performance for these machines (e.g. [2] and [24] both report that about half of all misses in important commercial workloads fall into this category). Prior solutions to this pressing performance problem have focused either on reducing the frequency of such misses, or on optimizing the coherence protocol to reduce the latency penalty of such misses. Optimizations that fall into the latter category include snooping protocols that avoid directory indirection to find the dirty block, reducing latency at the expense of additional address traffic (e.g. [24]), as well as schemes for dynamic self-invalidation (DSI) of dirty blocks, to avoid the latency cost of indirection in directory-based protocols (e.g. [19]). DSI proactively writes the dirty block back to memory, and allows a subsequent remote read to be satisfied from memory, usually with lower latency than if the request were forwarded to the writer.

To reduce the frequency of communication misses, researchers have proposed schemes that eliminate misses due to false sharing, update silent sharing (also known as update false sharing), and temporal silent sharing. These approaches are illustrated in Figure 1, in the context of increasingly aggressive attempts to increase the useful lifetime of a cache block that is written by a remote processor. In Figure 1(a), the remote write and local read occur to nonoverlapping portions of the block, resulting in false sharing, which can be detected and a miss avoided by schemes such as the one described in [12]. In Figure 1(b), the remote write turns out to be silent (i.e. the value written matches the value already existing at that memory location), and techniques described in [21] can be used to detect and eliminate the unnecessary communication miss. Extending store silence to the temporal

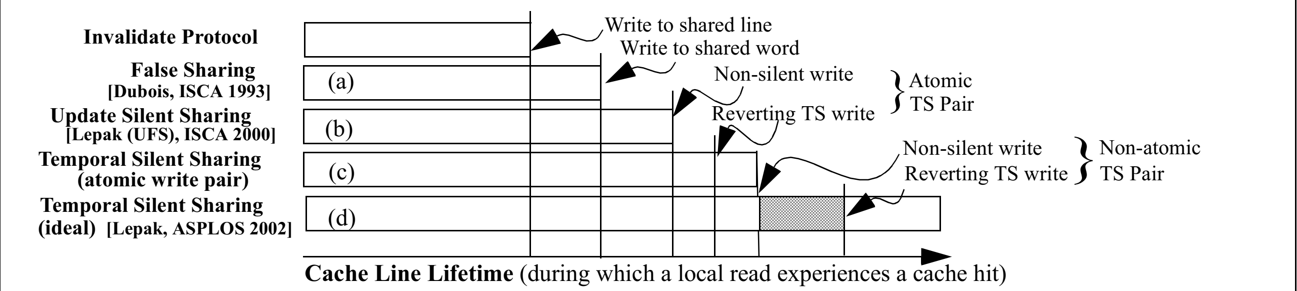

FIGURE 1. Extending the Lifetime of a Shared Cache Line. Lifetime is extended by avoiding writes to other parts of the word (false sharing), ignoring silent writes (update silent sharing), ignoring temporally silent atomic write pairs, and reinstating a line upon reversion due to a temporally silent write pair that is not atomic [22].

dimension, Figure 1(c) illustrates the case where the remote store is not silent, but is followed by an atomically collapsible second store (reverting write) where the cumulative effect of the two stores results in no architecturally visible change (i.e. the memory location reverts to the value it held before the first store). This property can be exploited to avoid coherence transactions caused by lock acquisitions and releases (speculative lock elision or SLE) as proposed by Rajwar and Goodman in [31]. However, doing so requires hardware support for guaranteeing that the *silent store pair* can in fact appear atomic to the rest of the system. Finally, Figure 1(d) illustrates the case where the silent store pair either cannot appear atomic or it is too cumbersome for the hardware to make it appear atomic. In such a case, the remote block must be temporarily invalidated between the two paired stores, but can be revalidated upon the reverting store. Techniques for exploiting this last case include the MESTI protocol, first described in [22].

All four of these cases can also be captured by speculating based on value at the consumer. Rather than conservatively waiting for the block to be fetched from the writer, the consumer can speculate that the stale value, likely still stored in the cache (albeit in invalid state), is probably correct, and use that stale value to continue execution. This is similar to the load value prediction first proposed in [23], but instead of relying on a separate predictor, uses stale cache contents as the source of value predictions. Of course, such speculative computation cannot be committed until the valid block has been fetched and its value compared against the predicted value. A trace-driven evaluation, along with a brief execution-driven study of the potential for such a scheme was presented in [15], but without a detailed description of hardware implementation and discussion/comparison with other methods of capturing the same benefit (through exploiting store value locality) in a unified simulation framework. Such a comparison, as performed in this work, reveals that speculation at the consumer (through load value prediction) is fundamentally less effective than removing the unnecessary communication at the producer by eliminating the invalidation effect of store operations (exploiting store value locality).

This paper presents a detailed, full-system, execution-driven evaluation of three of the value-based schemes (speculative lock elision or SLE [31], MESTI [22], and LVP [23]) in an effort to determine which of them is most effective at improving performance. Furthermore, we describe several improvements to MESTI that improve its efficacy and some enhancements to SLE that prevent it from causing considerable slowdowns with complex commercial workloads (prior work on SLE only evaluated scientific workloads with simple locking behavior); unfortunately, we were unable to achieve measurable speedup for SLE using commercial workloads. We also show that execution-driven evaluation is critically important for understanding the effects of these techniques, since the potential negative effects of coherence events can be difficult to gauge using trace-driven simulation. As an example, one might expect LVP to be the most effective technique, since it covers all the cases shown in Figure 1. However, LVP suffers from long verification latency due to inter-chip communication delay, and is in fact not able to reap as much benefit as MESTI, since even an aggressive (but still finite) processor window is not large enough to overlap the verification latency with other useful work. Finally, we show these techniques can be applied in concert, leading to speedups greater than what any technique can achieve in isolation.

The remainder of this paper is structured as follows: Section 2 describes the MESTI coherence protocol; Section 3 describes load value prediction with stale cache lines; Section 4 describes speculative lock elision; Section 5 presents a detailed performance evaluation, and Section 6 concludes the paper.

2 MESTI COHERENCE PROTOCOL

The MESTI protocol was introduced by Lepak et al. [22]. We summarize it briefly here to provide context for enhancements to the protocol presented in Section 2.3 and Section 2.4.

2.1 MESTI Protocol Review

The MESTI protocol (Figure 2) provides two essential functions for exploiting temporal silence—saving previous values of interest observed by remote processors and allowing re-installation of these cache lines into remote caches when temporal silence is detected. The version of the cache line which can be reverted to is kept in the *temporally invalid*, or T, state in remote processors. Temporally invalid lines can be reinstalled (transitioned to shared state) by sending a *validate* transaction. The protocol allows only a single previous system-visible value to be saved as a candidate for exhibiting temporal silence. In terms of mechanics, the modification from MESI is entering T state upon

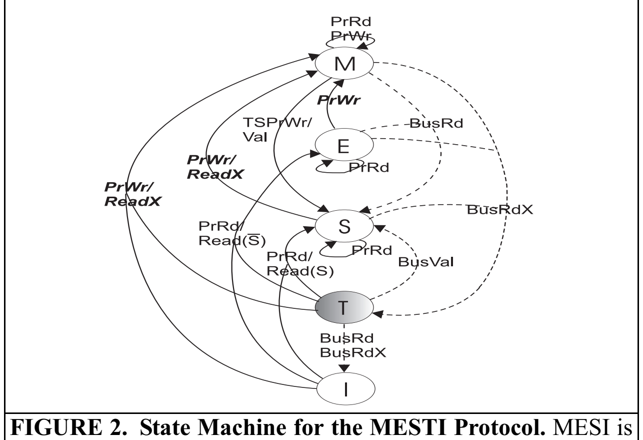

FIGURE 2. State Machine for the MESTI Protocol. MESI is augmented (using the notation from [10]) with temporal silence support by adding a temporally invalid (T) state. "Val" denotes MESTI's validate transaction used to communicate the occurrence of temporal silence against the previous globally visible value for a cache line; the italicized/bold PrWrs indicate where a stale version which can be reverted to is saved [22].

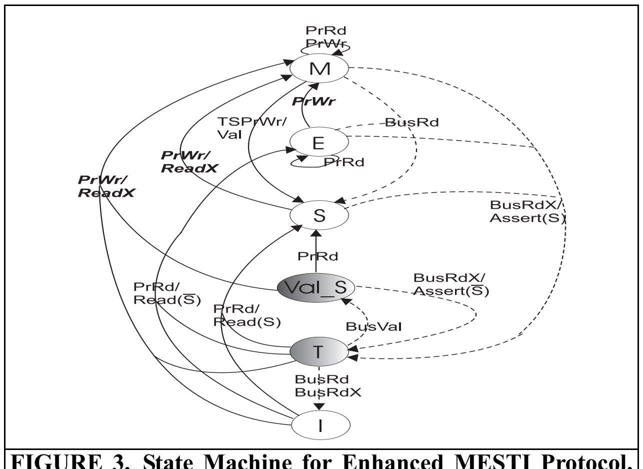

FIGURE 3. State Machine for Enhanced MESTI Protocol. We augment MESTI (Figure 2) using notation from Culler and Singh [10] to include the useful snoop response. Coherence events asserting the shared line are indicated.

receipt of an invalidate—saving the previously valid copy of the cache line, and returning the data to the shared state upon receipt of a validate, thus re-enabling hits on the cache line in remote processors.

2.2 Efficient Communication for MESTI

Each validate has the potential to eliminate multiple remote cache misses; therefore, MESTI has the potential to reduce coherence transactions. However, each validate also requires the validating processor to forego exclusive access to the cache line; thus, if a subsequent non-update silent store [21] occurs to a validated line, an upgrade is required. If a remote miss was not eliminated by the validate, less address traffic would have resulted, and store commit latency would have been improved, had exclusive ownership been maintained instead of broadcasting the validate. Lepak et al. showed that so-called *useless validates* are common, contributing a 10% to 100% increase in address transactions [22]. Therefore, we employ coherence prediction to minimize useless validates.

2.3 Using Distributed Communication to Implement Coherence Predictors

One method of reducing useless validates is the snoop-aware validate policy [22]. The snoop-aware validate policy simply collects snoop responses at each ReadX/Upgrade to determine whether any remote node had a valid copy of the cache line at the time of the request; if not, any subsequent validate due to a temporally silent store to the cache line on the owning processor is aborted. This mechanism eliminates many useless validates without sacrificing any opportunity; if no remote processor had a valid copy of the cache line at the intermediate value store, it is not possible for the cache line to be in T state, thus any validate is useless. The mechanism enabling this optimization is the *shared* snoop-response, which is used in machines implementing E state. In this context we have overloaded its use slightly, as it is only used for Read transactions in conventional MESI. This snoop-response mechanism enables a very simple form of distributed decision-making among processors when handling coherence requests since coherence actions taken by the requestor are affected by events in remote processors; in this case, valid data in remote caches.

Although the shared response has fixed behavior in conventional protocols, we envision using this distributed communication facility to implement predictive coherence mechanisms. For example, a remote processor may not assert the shared signal in response to an external Read request, even if it has a valid copy of the data. However, correct protocol operation requires it to treat the Read request as a ReadX, and invalidate the cache line, since the requestor may obtain the data in E state. If the remote processor suspects migratory sharing, this facility allows it to communicate that fact to a requesting processor implicitly.

We propose and evaluate a predictive coherence scheme for MESTI implemented using this mechanism. We add a single stable state to MESTI, called *Validate_Shared* which is entered upon receipt of a validate in T state. Therefore, it is semantically equivalent to S state for local requests; the only modification is any local request transitions a Validate_Shared cache line to S state. Upon receipt of an external ReadX/Upgrade transaction, we behave as specified in MESTI, except we abort assertion of the shared signal; all other states/transactions behave as normal. We call this response the *useful snoop response* because assertion of the shared line on an intermediate value store indicates a previous validate was useful and that it prevented a remote miss. The MESTI state machine for this enhancement is shown in Figure 3.

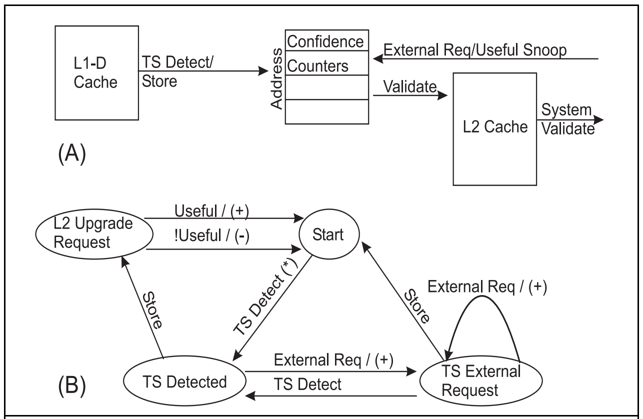

FIGURE 4. Address-Based Useful Validate Predictor. The predictor observes temporal silence detection and system snoop requests/responses to determine when a validate broadcast might prevent a remote cache miss. The system block diagram is shown in (A), while (B) indicates the Mealy state machine used to determine updates to confidence counters for each cache line. Transitions labeled (+) indicate confidence increments, (-) confidence decrements, and (*) the transition where a confidence value is read to determine system validate broadcast.

This simple behavior change allows a distributed prediction mechanism for useful validates. If a remote processor has not accessed the location since the previous validate, and therefore it is in the Validate_Shared state, it will not send a shared response upon subsequent intermediate value stores. Lack of the shared response indicates to the requestor that no remote copies were present. Hence it should abort future validates, as done in the Snoop-Aware Validate policy. If a remote processor has accessed the location, it will have transitioned to the Shared state, and will indicate a shared response, implying to the requestor that future validates may indeed be useful.

2.4 Coherence Prediction for MESTI

A block diagram of the predictor as well as a state diagram governing transitions is shown in Figure 4. The predictor observes temporal silence detection (TS Detect) and store requests from the cache and also external requests/responses from other processors. On each temporal silence detection, the predictor indexes into an array of confidence counters to determine whether a validate should be sent. If the confidence is above some threshold, a system validate is broadcast and the cache transitions to shared state; if below the threshold, a system validate is not broadcast and the cache returns to exclusive state. This check is indicated by (*) in Figure 4.

The state machine governing confidence counter updates is shown in Figure 4(B). At a high level, the state machine attempts to predict useful temporal silence. It does this by transitioning to the TS Detected state whenever temporal silence occurs. If an external request occurs for the cache line while it is temporally silent this indicates useful temporal silence, and thus a confidence

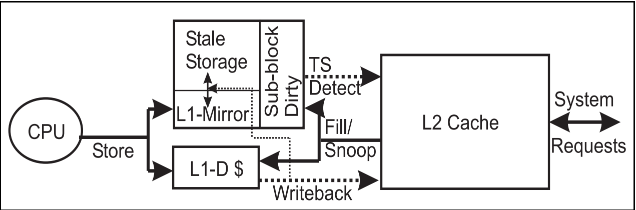

FIGURE 5. Illustration of an Efficient Stale Storage Mechanism.

increment, as indicated with the External Req arc. When a subsequent non-update silent store occurs in the TS Detected state, we use the useful snoop response to allow continuous training; we discuss this in detail shortly.

2.4.1 Exploiting the Useful Snoop Response to Improve Coherence Prediction in MESTI

A potential pitfall when designing this predictor is continually updating it once it starts eliminating remote misses with validates; once misses are not observed (because they are prevented) visibility of validate usefulness is lost. Put another way, it is easy to imagine training a sharing-predictor when misses to cache lines of interest are actually observed, but how is it trained when the misses no longer exist because they are actively eliminated? One can imagine periodically re-introducing the misses (by not sending validates for lines which exhibit temporal silence) to be sure remote processors are still benefiting from the validates, similar to the way hybrid update/invalidate protocols such as Dragon exit the update mode [10].

Instead, we can use the useful snoop response to enable continuous training. This response, sent at intermediate value store upgrade requests, is used to differentiate between useful and useless cases of temporal silence in the transitions from *TS Detected* to *L2 Upgrade Request* to *Start* (Figure 4). When the intermediate value store upgrade occurs, processors which accessed validated data will have transitioned to shared and assert the useful response (shared signal). If no processor has accessed the cache line since it was validated, it will be in either *invalid* or *Validate_Shared* state in remote caches. Therefore, the useful snoop response is not asserted and the predictor infers that a future validate is likely useless. Note that a separate state is needed for *L2 Upgrade Request* because the useful snoop response is generated after the store itself when the coherence agent has collected all snoop responses. Making each intermediate value store visible (Section 1) makes continuous training possible.

2.4.2 Limiting MESTI Coherence Predictor Storage

The predictor and confidence counters could be implemented as a separate structure; however, we advocate storing them directly in the L2 tag array, since the overhead is minor. Each predictor entry is only 2 bits for the state machine in Figure 4 plus confidence counters. When combined with the L2 tags, the predictor storage overhead is a function of L2 cache size and mapping; for example, a 16MB, 8-way associative, 64-byte line cache with 40 bit physical address has 5 bits of overhead (2 bits for predictor states plus 3 bits for confidence counters) / (~(6 baseline state bits) + (19 tag bits) for existing L2 cache tags) = 20% increase in L2 cache tag storage for the predictor. Since cache tags are a small fraction of the overall silicon area, the vast majority is the data array, the predictor storage overhead is likely less than 5%.

Furthermore, note that this prediction mechanism can be implemented entirely outside the processor core because it only observes physical memory addresses and standard coherence transactions; no PC or additional correlation information is used. Tuning the predictor cold miss confidence, confidence threshold, and confidence change values for various events was determined experimentally. We present results for initial confidence: 3, confidence threshold: 4, increment: 1, decrement: 1, and saturation value: 7, i.e. 3-4-1-1-7 in Section 5. Additional discussion on coherence prediction, tuning, and other predictors are not detailed here for the sake of brevity [20].

2.5 Efficiently Detecting Temporal Silence for MESTI

MESTI requires a mechanism to detect reversion of data to the previous version to enable validate broadcast. Lepak et al. [22] briefly discuss methods for implementing this *stale value storage*, and assume in previous work that an entire copy of cached data is available to detect all temporal silence immediately. Naively, this implies duplication of the entire memory hierarchy. An enhancement explored was to limit stale storage to some fraction of each cache line; although this policy works well, it still implies substantial data storage overhead. As significant extensions to this prior work, we make the stale storage overhead nearly negligible. For brevity, we do not discuss methods utilizing value summaries common in all modern memory hierarchies (ECC), and inclusive hierarchies, and provide this discussion elsewhere for the interested reader [20]. The results of these studies indicate that inclusive hierarchies can detect almost all useful cases of temporal silence for MESTI at only a slight L1-L2 interface bandwidth overhead. However, we show one of our most promising methods for detecting temporal silence. We discuss the proposed technique in the context of a writeback hierarchy, but it is extensible in a straight-forward way for write-through hierarchies. We assume that global coherence state is maintained at the L2 cache.

2.5.1 Adding Explicit Storage

A block diagram of the mechanism is shown in Figure 5. The basic principle of operation is simple: whenever the L1-D cache displaces a dirty line, the previous copy of the line (before it was initially dirtied from the previous version) is saved for future temporal silence detection. This is done in the stale storage shown in the figure. Note that obtaining the correct stale value on the first writeback requires reading data from the L2; the L1-D writeback data cannot be captured since the intermediate value is already stored within the cache line. To rectify this, the writeback can be converted into a read-write operation, doubling L1-L2 interface bandwidth. Another option, shown in the figure, is to add an explicit L1-Mirror which captures the correct stale value when the cache line is initially filled into the L1-D cache. If the cache line is subsequently modified and written back, the data from the L1-Mirror is copied to the stale storage and the dirty data is written in the L2, avoiding the L2 read-write sequence.

A slight complication arises in design of the L1-Mirror and its interaction with the stale storage and L2 cache. Namely, when a fill occurs, the L1-Mirror must correctly capture the temporal silence candidate data. The correct candidate, corresponding to the boldface-italic arcs in Figure 2, is the incoming data from the L2 if a previous writeback of an intermediate value has not occurred—otherwise, comes from the stale storage. Fortunately, this knowledge can be obtained from the L2; during the fill, the L2 simply indicates whether the line was previously written back, i.e. it is in M state at the L2, or whether the fill is a correct stale version (the labeled arcs in Figure 2). The L1-Mirror then either captures the L2 fill data or reads data from the stale storage. The datapaths between the L1-D cache and L2 cache behave as normal. The most up-to-date copy of the cache line always resides in either the L1-D cache or in the L2—thus external snoops need not access the stale storage or L1-Mirror to service requests.

This structure allows immediate temporal silence detection; each store simultaneously writes into the L1-D cache and also compares its value against the L1-Mirror leading to no validate delay. Temporal silence for an entire cache line is indicated by the NOR of all sub-block dirty bits, as shown in Figure 5.

Figure 6 shows the ability of this structure to capture useful

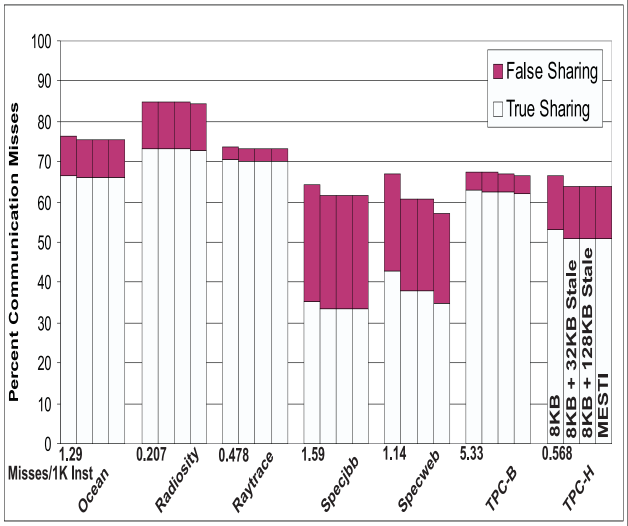

FIGURE 6. Communication Misses for Different Combinations of Stale Storage. Communication misses for an 8KB (4-way associative) L1-D cache (infinite L2) exploiting only inclusive hierarchies for temporal silence detection is compared to the same 8KB cache augmented with either 32KB or 128KB of stale storage. Results for MESTI with full stale storage (ideal case) are indicated in the right-most bar. The cost of the L1-Mirror is an additional 8KB in all cases.

temporally silent pairs under MESTI for a small 8KB, 4way, L1-D cache for stale storage capacities of 32KB and 128KB (for benchmarks described in Table 2 and a machine configuration similar to Table 1, details in [20])[1]. We see that both stale storage capacities are effective at capturing useful temporally silent pairs across all benchmarks. Note that comparisons are only performed against the L1-Mirror, and not the Stale Storage; thus accessing the L1-Mirror has similar delay to the L1-D cache since the configurations are identical. The Stale Storage can be slower since it is only written on L1-D cache writebacks, read on L1-D cache fills, and replacements from it cause no correctness issues because either the L1-D cache or the L2 maintains correct coherent data.

The space overhead is the size of the L1-Mirror plus Stale Storage. The smallest configuration shown (8KB L1-Mirror and 32KB Stale Storage) adds only 40KB/16MB=0.25% (Table 1) and achieves nearly all possible benefit. Therefore, all studies presented assume perfect temporal silence detection.

3 LOAD VALUE PREDICTION WITH INVALID CACHE LINES (LVP)

Seminal research by Lipasti and Shen [23] introduced value locality and value prediction to the academic community and promoted load value prediction (LVP) as a potential application of value locality. Since all silent store-related research builds on the foundation of value locality, it is fitting that we can return to this original work to provide another avenue to exploit TSS. More recently, Chang et al. examined various LVP methods (including tag-match invalid cache lines) to capture communication misses [6][15]. We discuss this related work in Section 6.

1. This structure cannot capture **all** temporally silent pairs--pairs longer than the lifetime captured in the L1-Mirror/Stale Storage cannot be detected; thus some communication misses above MESTI are shown.

3.1 LVP Captures Temporal Silent Sharing, False Sharing, and True Sharing

In Section 1, we discussed that TSS occurs when the value returned from the memory system on a miss is the same as the previous value observed by that processor. Therefore, a straightforward approach for capturing TSS is to simply use the values from tag-match invalid cache lines as value predictions. This approach has the desirable property of capturing all TSS misses, additionally all TSS false sharing misses, and a subset of TSS true sharing misses. Capturing TSS misses follows directly by definition. False sharing [14] [12] [22] misses are also be captured, since the referenced location has not in fact been written/changed. Capturing a subset of TSS true sharing misses is more subtle; the subset of interest are those cache lines which are truly shared, but there is sufficient time between a falsely shared demand miss and the eventual access which causes true sharing. The demand miss will be value verified to be correct and will have prefetched the rest of the cache line for true sharing.

3.2 Implementing LVP

When implementing LVP we must ensure value consistency [25]. Fortunately, mechanisms already present in many modern microprocessors for aggressive implementation of consistency models can be very simply extended for LVP [35] [13]. Our implementation relies on machine-squash recovery upon incorrect value speculation and LSQ snooping to ensure value consistency [25]; evaluation not detailed here [20] indicates machine squash recovery (compared to selective recovery) has negligible performance impact in our machine model.

Finally, we note that care must be taken in implementation considering multi-level memory hierarchies and controlling retirement within the microprocessor core. For example, it is desirable to enable tag-match invalid data to be used from any level in the cache hierarchy. The L1-L2 protocol should allow both the transfer of tag-match invalid data from the L2 for L1-D capacity misses to enable value speculation and a second transfer when data arrives from the system for prediction verification. In addition, an efficient mechanism for indicating when it is safe to retire a value predicted load is required.

Our solution to both these problems is to leave the protocol between levels in the memory hierarchy largely unmodified and simply deliver tag-match invalid data to the processor core without informing it that the data is speculative. This allows the processor core to be largely unaware that value prediction is occurring. We enable speculation control and mis-speculation signalling through an additional commit pointer which is logically part of the L1-D cache MSHRs. The MSHR contains additional state indicating that speculative data has been returned to the processor core, locations within the cache line which have been accessed by the core, and an identifier for the oldest operation in program order which has had speculative data returned. When data arrives from the memory system, the incoming data is compared against the tag-match invalid data. If the data matches for all locations which have delivered speculative data, the commit pointer is advanced; if it does not, the machine squashes at the oldest operation in program order attached to the MSHR. This mechanism is slightly pessimistic—only the first value mismatching operation must be squashed. However, this method allows each MSHR to track only a single operation index. To capture false sharing, tracking specific speculatively accessed locations is required; simply comparing the entire cache line returned from the system to the tag-match invalid data and signalling mis-speculation on any mismatch will unnecessarily signal value misprediction for all TSS false sharing misses.

Finally, note that setting the commit pointer correctly on any given cycle means finding the index of the oldest operation in program order with an outstanding value speculation across all MSHRs. Fortunately, this scanning of all MSHRs need only be performed each time any MSHR is accessed or updated which is only on L1-D cache misses and L1-D cache fills from the system.

4 SPECULATIVE LOCK ELISION (SLE)

Another method for exploiting temporally silent stores at the producer is Speculative Lock Elision (SLE) [31] [29]. The key insight of this work is that programming idioms intended to enforce mutual exclusion follow a common pattern. If the hardware can detect that a common idiom for mutual exclusion (such as a spin lock) is being executed, it infers that the programmer's desire is to enforce atomicity of operations between the lock acquire and release. SLE then elides the lock acquire and attempts to achieve atomicity of operations between the acquire and release using speculation and the coherence protocol. If atomicity can be guaranteed, it avoids acquiring the lock. This removes the artificial dependence on the lock variable serializing execution across processors and enforces the essential function the lock was intended to implement—non-conflicting access to data protected by the critical section. A related proposal is described by Martinez et al. [26].

SLE exploits temporal silence of the lock acquire (intermediate value store) and release (temporally silent store) as a key enabler to avoid making the intermediate value or temporally silent value visible outside the processor. If all memory operations within the lock acquire and release are atomic, the acquire/release pair appears to be an update silent store [21] to external observers and can therefore be collapsed into an atomic "silent store-pair" [31]. The benefits of this are two fold. First, misses to the lock variable itself are avoided since the lock is never acquired in the case of successful elision. Second, concurrent execution of non-conflicting atomic regions can also be achieved transparently in hardware.

We refer the reader to Rajwar's thesis for detailed discussion of SLE's mechanisms, guarantees of correct operation, and its detailed design [29]. We summarize the important differences in program behavior and SLE implementation in our simulation environment in subsequent sections.

4.1 Detecting Idioms for Elision

Since SLE relies on speculatively creating atomic regions, it must know when to ***start*** speculating, i.e, which store is the intermediate value store. Naively, any store is a candidate to begin the elision process. Attempting to elide every store has the principle shortcoming of leading to many false positives for elision candidates. A simple way to reduce the candidate set of intermediate value stores is to detect certain idioms [31] as high-confidence for elision. In previous work, only store-conditional operations were candidates for elision; more specifically, a pattern of load-locked/store-conditional operations, followed by a subsequent store to the same address [31]. This works well for the studies presented, since only user code is simulated for scientific benchmarks from the SPLASH-2 suite [34] where all synchronization code is user-supplied. Since store-conditionals were only used for lock acquires, this essentially provides instrumentation for their targeted references around critical sections.

We have observed, in full-system simulation environments under AIX/PowerPC (detailed discussion of the simulation environment in Section 5.2), that the load-locked/store-conditional part of the idiom which can begin speculation is much more prevalent. It is also used in contexts normally not good candidates for elision: clearing the reservation on context switches, atomic list insertion/deletion and increment/decrement, performing lock releases in addition to acquires, etc. The PC-based predictors used for elision candidates in previous work [29] may filter out many false positives. We discuss predictor effectiveness for our workloads in Section 4.2.3 and Section 5.

4.2 Implementing SLE

Essential aspects for implementing SLE are checkpointing of architected state when the elision process is started, providing atomic commit of all operations within a critical section when SLE succeeds, and detecting atomicity violations leading to elision failure or SLE restarts. Our implementation does not differ substantially from Rajwar's proposals [31] [29] in enabling atomic commit and detecting atomicity violations, so we neglect discussion of these aspects in our environment for brevity.

4.2.1 Speculation Buffering

SLE is a speculative approach. Therefore, many barriers to speculation apply (i.e. non-cacheable references, I/O accesses). However, changes in program flow, either due to microarchitectural state (e.g. branch predictions) or exceptions do not require speculation to fail. Also, SLE requires sufficient buffering and roll-back support within the processor core for speculative operations and speculative memory writes.

To provide speculation support Rajwar implements checkpointing along with explicit load snooping and store buffering outside the instruction window [29]. This allows capturing long distances between temporally silent pairs because speculation is only bounded by store buffer capacity (for speculative stores) and L1-D cache capacity (for loads). However, adding explicit checkpointing is a change to most existing out of order microarchitectures [16] [11] [9] [18] [28]. Therefore, we use existing speculation support, i.e. the reorder buffer, to perform SLE. Principally, this places additional constraints on the size of critical sections—they must be smaller than the ROB size.

In practice, we do not want to allow speculative critical sections as large as the entire ROB. Once elision begins, all instructions within the speculative critical section must complete in order to assure atomicity. This implies all operations within the critical section must be buffered inside the ROB until completion. Since a proper release may not be found (due to either imprecise elision idioms or insufficient buffering), allowing the entire ROB to fill with completed (but uncommitted) instructions in the hope a release will occur can sacrifice ILP. Therefore, we use a simple ROB threshold to bound speculative critical section size to some fraction of the ROB.

4.2.2 Datapath Issues with In-Core Buffering

Virtually all modern out-of-order microprocessors [28] [9] [11] [16] [18] impose datapath-related constraints to the OoO execution window. For example, incomplete forwarding within load-store sections (LSQ) [9] handled by allowing the execution window to drain will cause SLE failures since all values for speculative execution cannot be delivered by the datapath.[1] More importantly, many architected registers relating to process state, e.g. page table base address, are not renamed due to significant datapath complications or other micro-architecture-specific reasons. We call instructions writing such a piece of state *context serializing instructions*. When a context serializing instruction is executed, it drains all older instructions from the window, exe-

1. We observed insignificant (less than 1%) SLE failures due to LSQ forwarding constraints in our model.

cutes, and then allows subsequent instructions to be inserted into the OoO window. Obviously, context serializing instructions create a barrier to speculation. Virtually all critical sections encountered within the kernel and library routines in our full-system simulation environment using AIX v4.3.1 (Section 5.2) lead to context serialization due to an *isync* instruction [27] protecting lock acquire/release [8]. Of course, this would lead to failure for SLE for all kernel critical sections if handled naively. Instead of renaming such context-sensitive state, we implemented a mechanism which examines inserted instructions following *isyncs* within an SLE speculation region to surmise whether any context-sensitive state was accessed. In the case of unsafe SLE, we abort, otherwise SLE continues as normal. This mechanism captured virtually all kernel/library critical sections for the workloads we studied (Section 5.2).

4.2.3 Elision Confidence Prediction

We have made substantial effort at improving SLE's performance, since we observed that the simple restart threshold used in Rajwar's work [29] significantly degraded performance against the baseline machine for commercial workloads. We have implemented a more elaborate predictor which maintains hysteresis for given static instructions indicating successful elisions and changes in confidence given different SLE failure modes (idiom imprecision, SLE conflicts within critical sections, datapath constraints for atomic commit, etc.). All confidence update values were determined empirically.

Two issues with instruction-based prediction are: First, the load-locked/store-conditional pair which signals the start of the elision process can be used to implement other synchronization constructs (such as atomic list insertion, fetch-and-add, etc.). Second, and more fundamentally, few static instructions are participating in lock acquire/release since most or sometimes all locks are implemented with kernel-level functions [22]. Therefore, substantial interference in the predictor occurs between critical sections exhibiting different elision behavior, critical section length, etc. Therefore, the predictor cannot be made too conservative or too aggressive in disabling or enabling SLE for each static instruction (since it is the actual process of SLE which determines whether acquire idioms are amenable to SLE). A straightforward solution would be to use memory address-based prediction to uniquely identify critical sections. However, appropriately determining the speculation window after instructions have been inserted poses complication in state recovery mechanisms. This is true particularly for the checkpoint-based approach in Rajwar's thesis [29]—also for the in-core approach used here; the complications are described in detail elsewhere for brevity [20]. Restructured or instrumented kernel locking routines may have a large impact and is likely the best solution to avoid these problems, but we are not able to study this due to our use of a closed-source operating system (IBM AIX). More elaborate predictors may also provide leverage—however, we do not believe this is the best approach due to many possible interactions between the SLE algorithm, the processor core, and the coherence network as documented in Rajwar's thesis [29] and our own work available elsewhere [20].

5 METHOD COMPARISON AND PERFORMANCE EVALUATION

Before delving into detailed performance evaluation, we summarize key differences and attributes of the three methods exploiting TSS we explore. In our discussion, we use the term "speculative" to imply that a method requires facility for speculative execution and state recovery in the case of a mis-speculation.

5.1 Comparison of Methods

5.1.1 MESTI

The MESTI protocol is non-speculative and can be completely implemented outside the processor core. The coherence prediction mechanism requires only memory addresses for training and update. Furthermore, The predictor can be trained only on L2 state changes, allowing complete integration into the L2.

The principal drawback of MESTI is its reliance on explicit communication. Since MESTI makes both the intermediate value and temporally silent value visible for each temporally silent store, we must decide if broadcasting a validate will prevent a dynamic occurrence of TSS. Explicit communication can introduce additional coherence transactions both on the system coherence network and on-chip. Finally, explicit communication makes MESTI's ability to capture TSS dependent on attributes of the coherent interconnect; validates must reach a remote processor in a timely fashion (before a remote access) for communication misses to be successfully eliminated.

5.1.2 LVP

LVP is a speculative method for capturing TSS which exploits implicit communication, relying on speculation at consumers of the value as the means to implicitly "communicate" temporal silence. Because it is a consumer-based method (as opposed to MESTI and SLE which are producer-based) it can achieve ancillary benefits of capturing false sharing and a subset of true sharing by removing conservatism introduced by large coherence units.

The principal drawback of LVP is its consumer-based nature; the latency incurred to verify the prediction is partially exposed at remote processors. In the case of successful LVP, if no additional instruction-level parallelism (ILP) or memory-level parallelism (MLP) is exposed through correct early value delivery, we expect no performance benefit to result; the machine will simply stall waiting for data transfer to verify the prediction, similar to the base case where the load is a cache miss. We emphasize that this implies any evaluation of LVP without considering ILP/MLP effects, i.e. trace-based analysis (a major component of [6]), is inconclusive. Our own work [20] illustrates many common cases where LVP provides correct predictions, but will not expose additional ILP/MLP.

In the case of unsuccessful LVP, i.e. value misprediction, we incur penalties due to additional speculative execution and state recovery. In the base case these penalties are avoided because execution waits for data from the memory system before propagating it further in execution. Finally, temporally silent data must actually traverse the coherence protocol to verify predictions. In contrast, MESTI and SLE can completely eliminate value transfer through validate transactions and silent store-pair elision, respectively.

5.1.3 SLE

SLE is a speculative method which combines the producer-based and implicit communication attributes of MESTI and LVP, achieving the benefits of both. It can completely hide communication latency exposed to remote processors by eliminating invalidates for elided stores and also reduce coherence transactions in the system, as compared to MESTI, through the same means. In addition to eliminating TSS communication misses for idioms it can exploit, SLE can remove the serialization induced by conservative locking mechanisms in parallel programs; this cannot be achieved by either MESTI or LVP.

The principal drawback of SLE is its reliance on detecting idioms which are candidates for elision and requiring that tempo-

Table 1: Simulated Machine Parameters. Functional unit latencies are shown in parenthesis. L0-Cache latencies indicate one cycle address generation plus one cycle data delivery (1+1). L1 and L2 cache latencies are additive (i.e. L0-miss, L1-hit latency is 1+1+4=6 cycles, L0-miss, L1-miss, L2-hit latency is 1+1+4+15=21 cycles).

Attribute	Value
Fetch/Xlate/Decode/Issue/Commit	8/8/8/8/8
Pipeline Depth	6 stages
BTB/B-Pred/RAS	8K sets, 4-way/8K combining/32 entry
RUU/LSQ	256 entry/128 entry (micro-ops)
Integer	ALUS: 8 simple (1), 2 mul/div (3/12); 4 LD/ST
Floating Point	ALUS: 3 add/sub (4/4), 3 mul/div/fmac (4/4/4)
L0-Caches	I$: 64KB, 1-way, 64B lines (1+1); D$: 64KB, 1-way, 64B lines (1+1);
L1-Caches	I$: 512KB, 8-way, 64B lines (4); D$: 512KB, 8-way, 64B lines (4)
L2-Cache	64B-wide L1 interface, 1 cycle occupancy/txn; Unified: 16MB, 8-way, 64B lines (15)
Memory/Cache-to-Cache	Minimum latency: 400 cycles; 50 cycles occupancy/txn, crossbar
Address Network	Minimum latency: 200 cycles, 20 cycles occupancy/txn, bus (4-processor);
TLB	Hardware reload, 1-level, 2K sets, 2-way, 4Kpages
Memory Model	Sequential Consistency (MIPS R10K [35], [13])
SLE	In-core (RUU/LSQ speculation buffering, critical section size maximally 0.5*RUU/LSQ)
MOESTI	Instant temporal silence detection; L2-cache tags for validate prediction; O state is the same as in our bsaeline MOESI Gigaplane XB protocol [7].

rally silent pairs are atomic. All cases of temporal silence outside the idioms targeted, or which cannot be collapsed atomically, will be foregone; since MESTI and LVP do not focus on a particular idiom, arbitrary cases of TSS can be captured. As discussed throughout Section 4.2 many barriers to speculation may apply in practice, and effective idiom detection may be complicated. In addition, although SLE is a transparent hardware mechanism and correct function is guaranteed with imprecise idioms, this imprecision can lead to negative consequences. Coherence transactions are introduced in an attempt to create atomic regions, potentially slowing execution on remote processors and penalties for recovering from incorrect speculative execution apply. Rajwar's thesis indicates that choosing the restart threshold correctly for SLE was important to minimizing these negative effects [29]. Finally, we note that the temporally silent pair *distance* (dynamic program distance between the intermediate value store and temporally silent store [22]) capturable with SLE is limited by the speculative execution window—without adding additional, SLE-specific, checkpointing mechanisms this implies only temporally silent pairs shorter than the ROB size can be captured. MESTI and LVP are not directly constrained by this size, although MESTI requires adequate stale value storage to detect temporal silence [22].

5.2 Simulation Model and Parameters

We use the PHARMsim full-system, fully-integrated, multiprocessor simulation environment for all studies [4]. PHARMsim inherits its operating environment from SimOS-PPC [17], which is the PowerPC ISA [27] and AIX v4.3.1 runtime. The microprocessor core is similar to SimpleMP/SimpleScalar 3.0 [30] [3] with an additional translation stage added for instruction cracking of complex PowerPC instructions. We simulate 4-processor shared-memory, snoop-based, multiprocessor systems in our performance results. Additional abbreviated studies for 8-processor and 16-processor systems are available elsewhere [20]. Inclusive, write-back L0, L1, and L2 cache hierarchies are used. All user, library, and kernel code is directly executed by the performance model, including I/O via a coherent DMA agent. Precise resource configurations are given in Table 1. Functional correctness of PHARMsim is guaranteed via checking with a semantically unmodified SimOS-PPC functional simulator.[1]

1. The details of the functional validation are non-trivial and beyond the scope of this work. However, the validation assures the architected state observed by every committed instruction in both PHARMsim and SimOS-PPC is identical without passing any execution semantic information between simulators (with the exception of *stdcx/stwcx* success/failure status for SLE). Therefore, generated executions can be recreated with a semantically unmodified SimOS-PPC and are considered functionally correct. Validation is described in detail in Lepak's thesis [20].

Table 2: Basic Application Benchmark Characteristics. Instructions exclude the operating system idle loop. Update silent stores are indicated. Temporally silent stores are those captured with MESTI. IPC is across all processors. All lock-based data structures in the SPLASH-2 applications are padded to minimize coherence conflicts.

Program	Description	Instr.	Micro-Ops	Loads	Stores	US Stores	TS Stores	IPC
ocean	SPLASH-2 Ocean (258x258)	859M	984M	250M	75M	9.1M	1.67M	5.31
radiosity	SPLASH-2 Light (-room -ae 5000.0 -en 0.050 -bf 0.10)	2.39B	3.26B	947M	647M	137M	2.84M	7.00
raytrace	SPLASH-2 Raytracing (teapot)	418M	567M	124M	94M	18M	1.04M	3.61
specjbb	Commercial Server-Side Java Application [32] [5]	1.08B	1.91B	430M	209M	72M	10.8M	0.875
specweb	Commercial Web-Serving [5] [32]	3.0B	4.63B	1.24B	1.17B	270M	13.6M	3.86
tpc-b	OLTP benchmark [33] (20 clients, 1000 txns, in-mem DB2 v6.1)	468M	841M	159M	106M	60M	1.96M	1.49
tpc-h	Commercial Decision Support [33] (TPC-H query 12)	1.61B	3.18B	1.02B	842M	220M	38.2M	1.39

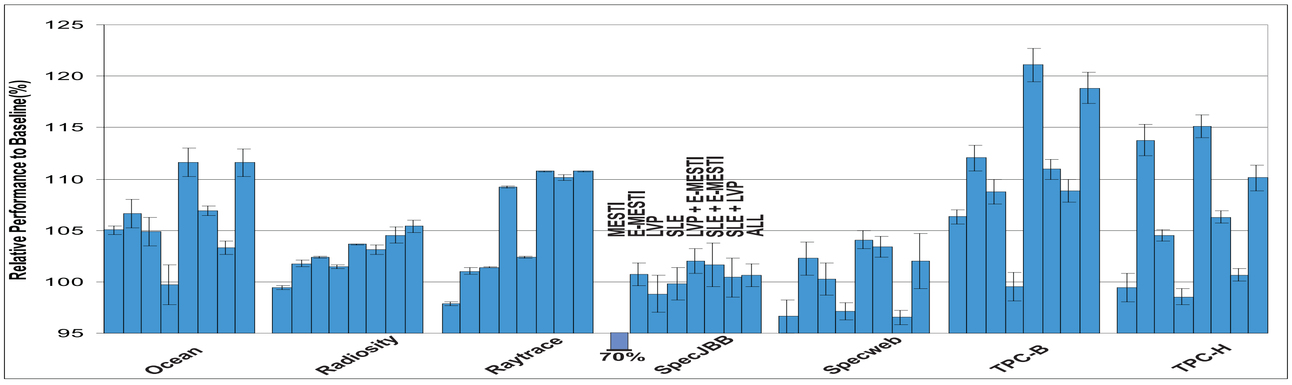

FIGURE 7. Performance Comparison of Application Benchmarks. All cases are normalized to Baseline performance with 95% confidence intervals for each benchmark also indicated**.**

5.3 Application Benchmark Results

We now compare the techniques with application benchmarks. Basic workload properties are shown in Table 2. We simulate three SPLASH-2 codes [34] and four commercial workloads with execution-driven simulation and measure performance using accepted statistical methods required for non-deterministic workloads [1].

Performance results for MESTI, LVP, and SLE, as well as all combinations of techniques, are presented in Figure 7. Address transactions are shown in Figure 8. Throughout all performance studies we observe the greatest performance sensitivity in *tpc-b* and *tpc-h* since these workloads have the highest L2-misses per instruction (Figure 8); many times an order of magnitude larger than the scientific workloads.

5.3.1 Techniques in Isolation

First, examining MESTI performance, we observe performance ranging from a 30% slowdown in *specjbb* to a 6.5% improvement in *tpc-b*. Adding coherence prediction (Section 2.4), which we call Enhanced-MESTI (E-MESTI), improves or maintains performance as compared to MESTI in all cases. This behavior is readily understood by examining Figure 8. *Radiosity* and all commercial workloads observe a substantial increase in address transactions over the baseline with MESTI; since the significant increase in validates is not offset by equivalent reductions in data transactions (Read/ReadX), obviously many validates with MESTI are useless. Coherence prediction successfully eliminates many of these useless validates, thus leading to improved performance (E-MESTI). In *ocean*, *radiosity*, *specweb*, and *tpc-b*, we see that while many useless validates are eliminated, a measurable increase in Read/ReadX transactions is observed for E-MESTI compared to MESTI. This indicates coherence prediction is also sacrificing some opportunity for sharing miss reduction. However, the foregone opportunity is offset by substantially reducing coherence transactions, leading to better relative performance for E-MESTI in all cases. In other results not detailed here, we have found that in *specweb* and *tpc-b* the sharing pattern is more complicated than the simple predictor can capture and lost opportunity is not due to predictor cold misses or capacity [20]. However, coherence prediction successfully eliminates many useless validates and maintains or improves performance, making E-MESTI more suitable for coherence bandwidth-limited environments. Performance improvement for E-MESTI strongly tracks data miss reduction (Read/ReadX) in the high miss-rate commercial workloads, indicating that E-MESTI effectively eliminates many TSS communication misses. Finally, we note that *specjbb* exhibits little performance sensitivity (except for a substantial performance loss in MESTI due to increased coherence traffic) even though it has a high L2 miss rate; this occurs because most misses in *specjbb* are capacity misses. None of the techniques presented provides additional leverage, as they target communication misses exclusively.

LVP is at best similar to E-MESTI in delivering performance for all workloads studied, and is in general less effective. We have discussed at length (Section 5.1) why this is plausible. Furthermore, Figure 8 shows that LVP cannot eliminate data transactions (Read/ReadX), as compared to E-MESTI and MESTI; in practice both data transactions and total coherence transactions increase with LVP due to additional wrong-path effects. However, LVP does achieve tangible performance benefit in many cases, i.e. *ocean*, *radiosity*, *tpc-b*, and t*pc-h* and only shows a measurable (although negligible) slowdown in *specjbb*.

SLE improves performance over the baseline in *radiosity* and *raytrace*, improving performance by 2% and 9%, respectively. In *radiosity*, the improvements are similar to those delivered by other methods. In *raytrace*, SLE achieves a measurable speedup beyond E-MESTI and LVP, indicating that it is exposing additional parallelism. In *ocean*, we observe a slowdown of 2%. Detailed examination revealed that *ocean* is most strongly affected by imprecision of the elision idiom (experiments with a 1K-entry RUU and 512-entry LSQ with 75% RUU/LSQ thresholds for SLE verified that insufficient buffering was not the primary factor for this workload).

Our SPLASH-2 benchmarks include initialization and parallel phases. The user-level locking routines within SPLASH-2 are amenable to SLE, since the load-locked/store-conditional pattern beginning elision is only used for locking routines. In *radiosity* and *raytrace* we have observed little operating system interference, thus the elision idiom is very precise and leads to few false positives for elision candidates. Within *ocean* we have observed substantial contribution from the operating system (predominantly during the initialization phase), leading to greater imprecision in the elision idiom. The result in *ocean* is mirrored across the commercial workloads. Performance in *tpc-b* and *specjbb* shows no meaningful difference; *specweb* and *tpc-h* show slight slowdowns of 3.0% and 1.5%. In all commercial workloads, substantial performance enhancement potential is sacrificed as compared to MESTI and LVP. Examining Figure 8, we see that SLE can reduce coherence transactions over baseline execution, as we have discussed previously, i.e. *radiosity* and *raytrace*. In *specweb*

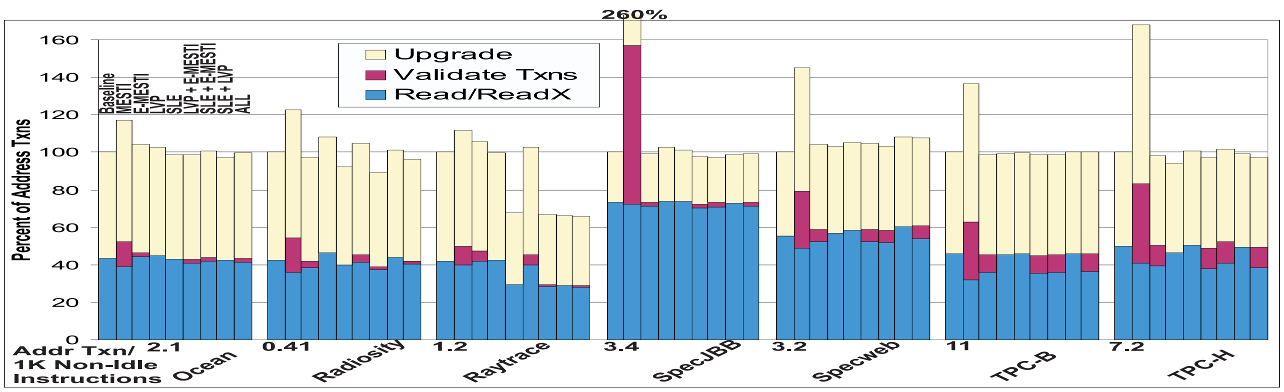

FIGURE 8. Address Transactions for Application Benchmarks. All cases are normalized to Baseline**.**

a slight increase is observed due to SLE failures/restarts (Read/ReadX) and exclusive prefetches necessary for forming atomic regions (Upgrade).

We have made substantial effort at improving SLE's performance for the commercial workloads, as described in Section 4.2.3. This improved performance for all commercial workloads; without these enhancements, all commercial workloads showed significant slowdowns (5-10%). Even with these enhancements, approximately 70% of load-locked/store-conditional pairs (elision candidates) which pass the predictor never encounter a release, even though only 25% of load-locked/store-conditional acquire idioms attempt elision. Practically, this implies that only about 8% (30% * 25%) of load-locked/store-conditional idioms are successfully elided. Note that this is not due to SLE conflicts in general, it is due to imprecision of the elision idiom. The load-locked/store-conditional pair which triggers the elision process is used to implement many other constructs than lock acquires; It can also be used for atomic list insertion, and other constructs. We have already discussed why designing more elaborate prediction mechanisms is either difficult, or likely not the best course of action to eliminate these drawbacks (Section 4.2). This implies that SLE, as currently proposed, is not a suitable transparent extension to existing systems; it can guarantee correct operation of such systems [29] but achieving robust performance appears more elusive.

5.3.2 Combined Techniques

Although there is substantial overlap in the references captured with each mechanism, nothing precludes combining the techniques (Section 5.1). Figure 7 and Figure 8 also show results for all combinations of the three techniques. Most noteworthy is the performance delivered when combining E-MESTI and LVP. Performance improvements, approximately equal to the sum of each method applied in isolation, are observed across all applications. This indicates that the set of references accelerated with these two techniques is disjoint—implying that LVP most effectively mitigates false sharing as opposed to TSS, and also underscores the point made in Section 3.2 that any study of LVP is inconclusive without considering ILP/MLP. With the parameters listed in Table 1, false sharing contributes 10-20% of communication misses in the scientific workloads, 20-30% in the commercial workloads.

When either E-MESTI and/or LVP are combined with SLE in the scientific workloads, we again observe nearly additive performance. Since LVP achieves benefit mostly from false sharing and SLE from true sharing and removing conservative locking, for combined SLE and LVP, this result is expected. Comparing E-MESTI and SLE combined, the overall result is most easily illustrated in *radiosity*. E-MESTI alone improves performance by 2.0%, SLE alone by 2.5%, but the combined techniques by only 3.0%. This indicates substantial overlap exists in references captured by each technique and re-illustrates Rajwar's results [31] that although SLE allows concurrent non-conflicting critical section execution, in fact, simply eliminating the lock transfer (exploiting temporal silence) is the primary benefit. However, in both *radiosity* and *raytrace*, combining SLE and E-MESTI improves performance over each method in isolation, reiterating results elsewhere [22] that not all TSS occurs in synchronization references and/or idioms readily exploited with SLE, even in scientific workloads with low operating system interaction. Finally, we note that enabling all three techniques provides the best performance in these workloads.

In the commercial workloads, combining SLE with E-MESTI and/or LVP actually hinders performance, as we expect from the results from SLE in isolation on these workloads. Examining Figure 8, we do not observe substantial negative interference from coherence traffic attempting to form atomic regions (although measurable at 1-2% in all workloads), further indicating that the primary culprit is imprecision of the elision idiom and negative interaction with the processor core (Section 4.2.1 and Section 5.1.3).

6 SUMMARY AND FUTURE WORK

In this paper we have discussed three methods of capturing temporal silent sharing: the MESTI protocol, load value prediction (LVP), and speculative lock elision (SLE). We contribute a novel predictor mechanism to deliver robust performance for MESTI (E-MESTI), space-efficient and timely mechanisms for detecting temporal silence, and a detailed discussion and performance evaluation for LVP using a fully-integrated, full-system simulation environment capable of running commercial workloads. Chang et al. discussed LVP (with additional prediction schemes) in other work, with an emphasis on trace-based evaluation and selected execution-driven results [6][15]. We have shown that such evaluations do not provide a complete picture of expected performance because they cannot properly account for the degree to which verification latency can be overlapped with other useful work; we find that LVP is often not able to provide much benefit given our machine model, and expect that the trend towards increased processor frequency without a corresponding improvement in off-chip latencies will only exacerbate this problem. We compare our new proposals to previous MESTI and SLE proposals with detailed discussion and contribute an evaluation which combines full-system simulation, out of order processor models, and commercial workloads. We have also shown that eliminating communication misses by removing the invalidation effect of stores, as opposed to relying on load value speculation at the consumer, is more effective, even though value speculation can theoretically capture more misses.

In our scientific and commercial application workloads, we observe performance improvements ranging from 1.0% to 14% for an enhanced version of MESTI, -1.0% to 9.0% for LVP, and -3.0% to 9.0% for SLE. We also show that all techniques can be combined in scientific workloads to achieve the best possible performance. However, due to SLE's negative interaction with the processor core in commercial workloads, only enhanced MESTI and LVP should be combined with existing codes and idiom detection techniques because they can capture arbitrary temporally silent idioms. Enhanced MESTI combined with LVP achieves 2.0% to 21% performance improvement in these workloads. We provide discussion of the observed behavior in all cases. Our studies imply that while SLE can theoretically eliminate communication misses and enable concurrent execution of non-conflicting critical sections, accurate detection of elision idioms is vitally important for delivering robust performance and appears difficult for existing codes.

We have explored these methods in a snoop-based multiprocessor system. MESTI, LVP, and SLE can be implemented directly in directory-based systems [31] [20]. However, mechanisms for coherence prediction in MESTI relying on the useful snoop response may need modification since generating this response is more complicated or may be not be feasible [20]. Completely different predictors may be required in such systems. Further exploring SLE idioms and operating system instrumentation for commercial workloads may improve its potential in these applications. Finally, we exploit only update silence and temporal silence in this work; more elaborate value communication schemes leveraging ideas presented for MESTI may provide other fruitful results.

References

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th Annual International Symposium on High Performance Computer Architecture*, 2003.

[2] L. A. Barroso, K. Gharachorloo, and F. E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[3] D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin Computer Sciences, 1997.

[4] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. Proceedings of Computer Architecture Evaluation using Commercial Workloads (CAECW-02), February 2002.

[5] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural characterization of Java TPC-W. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 229–240, Monterrey, Mexico, January 2001.

[6] J. Chang, J. Huh, R. Desikan, D. C. Burger, and G. S. Sohi. Using coherence value speculation to improve multiprocessor performance. In *Proceedings of the First Value-Prediction Workshop in conjunction with ISCA-2003*, 2003.

[7] A. Charlesworth, A. Phelps aand R. Williams, and G. Gilbert. Gigaplane-XB: Extending the ultra enterprise family. In *Proceedings of the International Symposium on High Performance Interconnects V*, August 1997.

[8] IBM Corporation. AIX v4.3 online documentation. http://ncsp.upenn.edu/aix4.3html/, 2002.

[9] Intel Corporation. *Intel Pentium 4 Processor Optimization Reference Manual*. Intel Corporation, Santa Clara, CA, 2000.

[10] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1999.

[11] K. Diefendorff. K7 challenges intel. *Microprocessor Report*, 12(7), October 1998.

[12] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *20th Annual International Symposium on Computer Architecture*, May 1993.

[13] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[14] J. R. Goodman and P. J. Woest. The wisconsin multicube: A new large-scale cache coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988.

[15] J. Huh, J. Chang, D. C. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *Proceedings of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004.

[16] J. Keller. The 21264: A superscalar Alpha microprocessor with out-of-order execution. In *Proceedings of the Microprocessor Forum*, October 1996.

[17] T. Keller, A. M. Maynard, R. Simpson, and P. Bohrer. SimOS-PPC full system simulator. http://www.cs.utexas.edu/users/cart/simOS.

[18] J. Lachman and J. Michael Hill. A 500mhz 1.5mb cache with on-chip CPU. In *Proceedings of ISSCC-1999*, February 1999.

[19] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, 1995.

[20] K. Lepak. *Exploring, Defining, and Exploiting Recent Store Value Locality*. PhD thesis, University of Wisconsin, Department of Electrical Engineering, 2003.

[21] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 182–191, Vancouver, B.C., Canada, June 2000.

[22] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *Proceedings of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 30–41, October 2002.

[23] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.

[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. *ACM SIG-PLAN Notices*, 35(11):25–36, November 2000.

[25] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proceedings of MICRO-34*, December 2001.

[26] J. F. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applicatioons. In *Proceedings of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[27] C. May, E. Silha, R. Simpson, and H. Warren. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., 1994.

[28] C. Moore. POWER4 system microarchitecture. In *Proceedings of the Microprocessor Forum*, October 2000.

[29] R. Rajwar. *Speculation-Based Techniques for Transactional Lock-Free Execution of Lock-Based Programs*. PhD thesis, University of Wisconsin, 2002.

[30] R. Rajwar and J. R. Goodman. SimpleMP multiprocessor simulator. Personal communication, 2000.

[31] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, December 2001.

[32] Systems Performance Evaluation Cooperative. SPEC benchmarks. http://www.spec.org.

[33] Transaction Processing Performance Council. TPC benchmarks. http://www.tpc.org.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[35] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.