Edge Chasing Delayed Consistency: Pushing the Limits of Weak Memory Models

Harold W. Cain
IBM T. J. Watson Research Center
Yorktown Heights, NY
tcain@us.ibm.com

Mikko H. Lipasti
University of Wisconsin
Madison, WI
mikko@engr.wisc.edu

Abstract

In shared memory multiprocessors utilizing invalidation-based coherence protocols, cache misses caused by inter-processor communication are a dominant source of processor stall cycles for many applications. We explore a novel coherence protocol implementation called edge-chasing delayed consistency (ECDC) that mitigates some of the performance degradation caused by this class of misses. Edge-chasing delayed consistency allows a processor to non-speculatively continue reading a cache line after receiving an invalidation from another core, without changing the consistency model offered to programmers. While the idea of using stale data for as long as possible is enticing, our study shows that the benefits of such delay are small, and that the majority of these delayed invalidation benefits come from mitigating the false sharing problem, rather than any tolerance of races or an application's ability to consume stale data in a productive manner.

Categories and Subject Descriptors B.3.2 [*Design Styles*]: Cache memories

1. Introduction

In shared memory multiprocessors utilizing invalidation-based coherence protocols, cache misses caused by inter-processor communication are a dominant source of processor stall cycles for many applications. Consequently, modern processors spend a significant fraction of their time sitting idle, waiting for a memory reference that could not be serviced by its local cache hierarchy and instead must be transmitted by a more distant source. In invalidation-based coherence protocols, when one processor is writing a particular memory location, that cache line is removed from the caches of other processors. Should those processors subsequently access the location, their access will incur cache misses and most likely result in processor stalls. Consequently, cache misses due to inter-processor communication significantly degrade performance for many parallel applications. The objective of this work is to enhance the performance of shared-memory multiprocessor systems by reducing the negative impact of coherence misses.

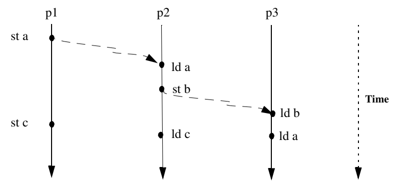

Figure 1. Causal dependencies through shared-memory.

Given a system composed of multiple processes where each process performs a sequence of events including inter-process send and receive operations, Lamport defined the causality relation which specifies the necessary order of events in the system [18]. The causality relation states that the order in which events at one process become observable to other processes should reflect the sequence in which they occur within each process. Informally, in a shared-memory multiprocessor, when one processor reads a memory location that was previously written by another processor, or when one processor overwrites a memory location that has been previously read or written by other processors, that processor becomes causally dependent upon those prior read and write operations performed by other processors. Such causal dependencies are transitive in nature, as illustrated in Figure 1 in the context of a system consisting of three processors (*p1*, *p2*, *p3*) and three memory locations (*a*, *b*, *c*). Processor *p1* initially performs a store to memory location *a*, and *p2* subsequently reads the newly written value. Processor *p2* is now causally dependent upon *p1*'s store to a and those instructions on which *p1*'s store to *a* causally depend. Processor *p2* subsequently writes location *b*, which is then read by *p3*. When *p3* performs a load to location *a*, it is already causally dependent upon *p1*'s store to a transitively through the memory location *b*, and must therefore read the value written by *p1*. Two events are said to be concurrent if neither event is causally dependent upon the other. When *p2* executes its *ld c*, it is not already causally dependent upon *p1*'s store to *c*. Therefore, *p2*'s load to *c* may correctly return either the value written by *p1* or the value that existed at *c* prior to *p1*'s write. It is this type of ambiguity that the ECDC protocol exploits. A formal definition of the constraint graph when applied to the weakly ordered memory model can be found in prior work [5].

2. Edge-Chasing Delayed Consistency: A New Implementation of Weak Ordering

In an invalidation-based coherence protocol, a cache line may be invalidated from the cache but the previous copy of the data will remain cache-resident until a subsequent cache miss to that set; the line is marked invalid, but the tag-match logic will indicate a match. In many instances, this data may be useful to the processor. If a cache controller could identify those situations in which it is correct to use the stale data, it could return the stale data non-speculatively rather than stall the processor. This will reduce the latency observed by the processor reading the data, but can also aid the processor that currently has a modified copy of the data. If a new copy is not requested by the reader, then the writing processor retains an exclusive copy and can continue writing the line without sending an upgrade message.

In this paper, we describe an implementation of weak ordering called edge-chasing delayed consistency (ECDC). ECDC is a hardware mechanism that identifies stale lines that can be used non-speculatively, while continuing to provide a coherent and consistent shared memory image to software. Our ECDC implementation improves upon prior versions of delayed invalidation by extending the lifetime of stale cache lines beyond the execution of memory barrier or synchronization instructions by a processor. By detecting cycles in the constraint graph, ECDC allows the use of stale data until a processor becomes causally dependent upon the write that caused the line to become stale. ECDC is not a new programming model; it is simply a new way of implementing the existing weak ordering supported by the POWER and Arm instruction sets. The principles behind ECDC may be applied to other models, particularly weaker models such as release consistency.

We begin our discussion of ECDC in with a presentation of programming paradigms and microarchitectural artifacts illustrating those scenarios in which it is useful for a processor to continue using stale data after it has been invalidated. In Section 2.2, we describe how the constraint graph representation can be used to identify those cache lines that can safely be read when stale. In Section 2.3, we present a conceptual description of the ECDC protocol, which maintains the constraint graph in a distributed fashion, allowing a processor to use stale lines from it. In Section 4, we present a detailed evaluation of the ECDC protocol. We conclude with a discussion of prior work in Section 4.

2.1 Applications of Delayed Consistency

It may not be immediately obvious why it would ever be useful to continue using a cache line after it has been invalidated. The programmer updated that data for a reason, right? If she intended to communicate new data from one thread to another, then why would it ever be useful to delay that communication? Of course there are some cases where using stale data will not be useful, even though it may be safe with respect to the consistency model. For example, if a processor acquiring a lock continues to observe the held value of the lock after it has been released, then its acquire will be delayed, reducing performance. However, there are other cases where it does not matter whether the reader observes the old value or the new value; it is more important that the reader reads either of them quickly than wait on the newer value. In the next two subsections, we present examples of applications in which the use of truly shared stale data will improve performance: linked data structures shared among threads, and data-race tolerant iterative convergent algorithms.

Due to false sharing [13] and silent sharing [22], there are also instances in which a line has been invalidated but subsequent loads to that line will read the same value regardless of whether the stale data is returned or the new copy is fetched. This avoidable communication represents another opportunity to benefit from delayed consistency.

Linked data structures

If a shared data structure is a source of contention in a parallel application, elaborate locking schemes are frequently used to maximize concurrent access by readers and writers. Some algorithms allow readers of a linked data structure to continue to traverse the structure despite the presence of one or more concurrent writers. In such algorithms, a delayed invalidation mechanism should provide performance benefit by shielding a processor traversing the data structure from observing (and stalling) to read newly inserted nodes.

For example, a lock-free list insertion occurs in two steps, in which a new node is inserted between two existing nodes. As in any list insertion, the new node's next pointer is first set to the address of the subsequent node. In the second step, a compare and swap (CAS) operation is performed, replacing a next pointer so that it now points to the new node. If the CAS succeeds, then the new node has been successfully inserted and the operation is complete. A CAS failure indicates that another writer has either deleted prev or inserted a new node between prev and cur, in which case the insertion process must restart. If concurrent readers are traversing the list, in the absence of any other synchronization such readers may continue to read the pre-update version of the list. We present a microbenchmark study of the performance of ECDC running a lock-free list manipulation algorithm of this type in Section 4.

Asynchronous communication and convergent iterative algorithms

Delayed invalidation should also be useful as a method of implementing asynchronous communication in shared memory multiprocessors. Some applications benefit from the ability to read certain memory locations without caring whether or not the read returns a new version of the data or a previous version. Implementing this type of communication is difficult using current instruction sets because they do not support any kind of ”don’t care” loads. A load reads the newest data, wherever it exists in the system. Using non-binding prefetches is also difficult, because the prefetch must be timed perfectly to return the data before the binding load that subsequently reads the data is executed.

Convergent iterative algorithms are one class of algorithm that use this model of communication, which typically a shared data structure representing the current state of the solution. Although barrier synchronization may be performed between iterations, the shared copy of the solution is often accessed without synchronization. After a number of iterations, the application converges on a solution. Algorithms of this type include a plethora of parallel equation solvers, sparse matrix factorization (e.g. cholesky from SPLASH2 [31]), and many parallel genetic algorithms [29].

False sharing and silent sharing

False sharing is an artifact of the coherence granularity being larger than the smallest addressable unit of memory [13]. A processor *pwriter* may write some portion of a cache line, invalidating that line from another processor *preader*’s cache, and cause a miss at *preader* even if *preader* never subsequently touches the written parts of the line. As will be shown in the results section, there is a significant amount of false sharing in some of the workloads studied here, creating communication that we would like to avoid.

Lepak and Lipasti identified another source of unnecessary communication, caused by writes that either overwrite a value with the value already resident at that memory location [23], or revert a location’s value to a value that previously existed at that location [24]. Because a read to the stale version of a silently written line will consume the same value that would be consumed if the new data were fetched, delayed consistency can reduce the performance impact of this class of cache misses by using the stale data rather than waiting for the cache miss to return. Lepak found that between 18% and 44% of all coherence misses were attributable to silent sharing across a set of benchmarks [22].

Delayed consistency protocols can mitigate the performance impact of both false sharing and silent sharing misses. However, not all of these misses will be avoidable. If a processor is already causally dependent upon the write that invalidated a cache line, then it can no longer use the stale line in the ECDC protocol, even if the line is stale due to false sharing or silent sharing.

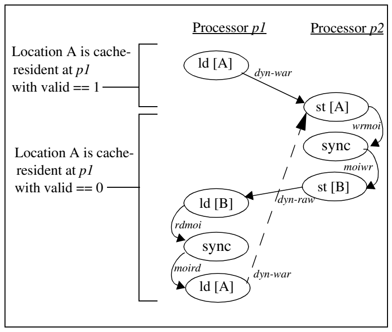

Figure 2. A necessary coherence miss

2.2 Identifying Usable Stale Data

Assuming that we would like to allow a processor to use stale data when possible, how can we identify those lines for which it is safe? Using the constraint graph, we can identify instances of communication between processors by examining the source and destination processors of each edge. Each read-after-write (RAW), write-after-write (WAW), or write-after-read (WAR) edge whose two endpoint instructions were executed by different processors equates to a single miss or upgrade between processors in an invalidation-based coherence protocol. Inter-processor RAW edges correspond to a read miss that is satisfied by a dirty copy of the memory location residing in another processor’s cache. Similarly, inter-processor WAW edges correspond to write misses satisfied by remote processors. Interprocessor WAR edges correspond to writes that result in either a miss or an upgrade message between processors.

Given a coherence miss caused by a load instruction, we can determine whether or not that miss is avoidable using the constraint graph based on the following criterion: a RAW edge *e* emanating from writer node *w* and connecting to reader node *r* is necessary if there exists a directed path in the constraint graph from *w* to *r* that does not include edge *e*. This observation follows from Landin’s proof that if a constraint graph is acyclic then the execution corresponding to that constraint graph is correct [19]. If *e* is deemed avoidable, then we are essentially transforming the RAW edge from *w* to *r* into a WAR edge from *r* to *w*. If there is already a directed path from *w* to *r*, this new WAR edge would create a cycle in the graph, and would thus be incorrect. If there is no directed path from *w* to *r*, then WAR edge cannot create a cycle, and the coherence miss is unnecessary.

An illustration of a necessary coherence miss under sequential consistency is shown in Figure 2. In this example, processor *p1* is about to perform its second load to cache line *A*, but the cache line containing A has been invalidated. In order to determine whether or not *p1* can avoid this cache miss, we look at the constraint graph node that would provide the value to the miss (the ”W” in RAW), in this case processor *p2*’s store to A. We would like to use the stale value from the cache, thus creating a WAR edge from *p1*’s load A to *p2*’s

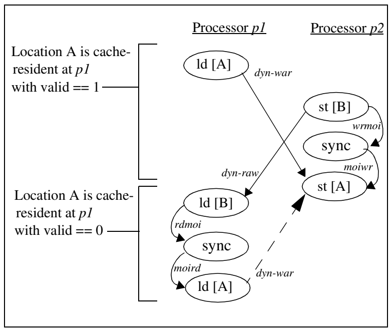

Figure 3. An unnecessary coherence miss

store *A* (indicated by the dotted arrow). However, if there already exists a directed path from this store node to the load miss node, then we know that this miss is necessary, because the load is already transitively causally dependent upon the store. As we can see in the figure, a directed path already exists from *p2*'s store *A* to *p1*'s load *A* through a RAW dependence on memory location *B*. If this path did not exist, p1 could safely use the stale value of *A*. However, because it does exist p1 must not use the stale value, thus the dotted WAR edge must be transformed to a RAW edge, eliminating the cycle.

Figure 3 illustrates a similar code segment, however in this case the load miss to location A by *p1* is avoidable. The miss is avoidable because we can create a legal schedule of operations such that the load miss will return the old value of *A*. In this example, *p2* performs the stores to memory location *A* and *B* in reverse order. Consequently, at the time of *p1*'s load miss, there is not already a directed path from *p2*'s store of *A* to *p1*'s load of *A*, therefore the WAR edge caused by using the stale value does not create a cycle, and this coherence miss is avoidable.

Speculative mechanisms might guess that it is safe for *p1* to use the stale data in *A* because the store to *A* by *p2* may have been a silent store or may have been to a different word within the same cache line. These mechanisms require a verification step to ensure that p1 did indeed load the correct value. Given the long latencies associated with fetching data from another processor's cache, this verification operation will most likely stall the processor. However, using the constraint graph, we can detect cases where it is safe to use the stale value, without the need for verification.

2.3 Edge-Chasing Delayed Consistency: A Conceptual Description

In this subsection, we describe the concepts behind an implementation of the weak ordering memory model called edge-chasing delayed consistency. In order to provide a clear description of this new caching algorithm, we separate the idea from the implementation and present a conceptual description in this subsection, not subject to any hardware constraints. A full description of the protocol implemented in our simulator and evaluated can be found in prior work [5].

Edge-chasing delayed consistency derives its name from a class of deadlock detection algorithms proposed for distributed database systems [6][17]. Processes in such systems can optimistically acquire and release locks as desired (i.e. do not follow rigid deadlock-free locking disciplines), and in the event of deadlock, abort one of the transactions participating in the deadlock, thus freeing that transaction's held locks and allowing the other transactions to proceed. Many algorithms perform detection through the construction of a waits-for-graph (WFG), a directed graph whose nodes correspond to processes, and whose edges represent the dependencies among processes (specifying which processes "wait-for" which other processes). For example, if a process A is blocked attempting to acquire a lock held by process B, there will be an edge from A to B in the WFG. A deadlock can be detected by testing the WFG for a cycle; if a cycle exists, then there must be a cyclic dependence of resources held by processes.

Edge-chasing algorithms detect cycles in the WFG in a distributed fashion through the propagation of special messages called probes communicated along the edges of the graph. When a process suspects the existence of a deadlock due to a timeout, it creates a probe and sends the probe to the process on which it waits. The recipient of the probe forwards the message on to the process on which it waits. The reception of a probe created by the receiving process indicates that the process is part of a cycle in the graph, causing the process to abort.

The problem of maintaining consistency in a shared-memory multiprocessor resembles the deadlock detection problem. Instead of checking the WFG for cycles, we will check the constraint graph for cycles using a similar mechanism. Keep in mind that in neither the deadlock detection scenario nor the memory consistency scenario do we need to explicitly construct or communicate the entire graph. Instead, the occurrence of a cycle can be inferred by the receipt of a locally created probe.

At a high level, ECDC works as follows: every write operation that invalidates a cache line from a remote cache initiates the creation of a probe. A probe is a globally unique identifier that is passed from processor to processor at the occurrence of certain events. A copy of this probe is kept with the stale copy of the invalidated cache line. The stale line may be used until the line's associated probe is received from another source.

When communicating with other processors through loads and stores to shared memory, the creator of a probe ensures that stale copies are not used incorrectly by passing the probe on to the other processors whenever the other processor's load or store will follow the probe-initiating write in the constraint graph. For example, after invalidating a remote cache line *c* and creating probe *pb*, processor *pwriter* writes a different memory location that is subsequently read by another processor *preader*. When *pwriter* sends the new data to *preader*, it also sends probe *pb*, because *preader* is now causally dependent upon processor *pwriter*'s write, meaning that if *preader* subsequently reads cache line *c*, it should observe the new copy of the line. By passing probes only along edges in the constraint graph, ECDC ensures stale data will remain useful as long as possible: until the processor using the data becomes causally dependent upon a newer version of the stale line.

In order to support this communication, the ECDC protocol maintains sets of probes for each processor and memory location, indicating the writes on which they causally depend. When a coherence message is sent, the probe set corresponding to that memory location is sometimes attached to an outgoing message. When a coherence message is received, the incoming probe set is added to a per-processor probe set and the memory location's probe set. A proposed hardware implementation of ECDC is described in full detail elsewhere, which optimizes the design to minimize probe overhead while approaching the same benefits of an idealized ECDC implementation[5].

3. Experimental Evaluation of Edge-Chasing Delayed Consistency

We have implemented two forms of the the ECDC protocol in a cycle-accurate full-system simulator: 1) an unlimited resource version which assumes an infinite namespace for probe identifiers and infinite storage area for maintaining probes at each cache, and 2) a realizable version of the protocol which minimizes the sizes of such meta-data. In this section, we present a detailed performance evaluation of unlimited resource version of the protocol. Additional data, including the description and evaluation of the limited resource variant can be found elsewhere[5].

We compare the ECDC protocol to a baseline machine utilizing a conventional directory coherence protocol, based on the SGI Origin [20], whose configuration is detailed in Section 3.1. In Section 3.2, we characterize the relevant behavior of coherence misses across a set of scientific and commercial applications to gauge the opportunity for performance gains from ECDC and to provide insight into the subsequent evaluation. In Section 4, we evaluate the performance of the ECDC protocol.

3.1 Machine Configuration

Table 1 describes the machine configuration used for these experiments. We use a four-processor baseline machine, with an interconnect topology (and latencies and bandwidths) based on the Alpha 21364 network [27]. We replace the 21364's dynamic routing protocol with a simpler static dimension-ordered routing mechanism.

3.2 Coherence Miss Characterization

Figure 4 shows the number of misses per 1000 committed instructions for a set of parallel applications. Each bar is broken into its cold, coherence, and capacity/conflict components [14]. The top of each bar additionally includes upgrade

Table 1. Simulated system configuration.

Out-of-order execution	5.0 GHZ, 15-stage 8-wide pipeline, 256 entry reorder buffer 128 entry load/store queue, 32 entry issue queue store-set predictor with 4k entry SSIT and 128 entry LFST
Functional Units (latency)	8 integer ALUs (1), 3 integer MULT/DIV (3/12) 4 floating point ALUs (4), 4 floating point MULT/DIV (4, 4), 4 L1D ports
Front-end	fetch stops at first taken branch in cycle combined bimodal (16k entry)/gshare (16k entry) with selector (16k entry) bpred 64 entry RAS, 8k entry 4-way BTB
Cache hierarchy (latency)	32k direct-mapped IL1 (1), 32k direct-mapped DL1 (1) 64 entry write buffer 512k 8-way DL2 (7), 512k 8-way IL2 (7) Unified 16MB 8-way L3 (15), 128 byte cache lines 2k entry 2-way ITLB, 2k entry 2-way DTLB. Stride-based prefetcher modeled after IBM Power4
Interconnect/Memory	2-D torus static dimension order routed interconnect. 15 ns (60 cycle) per link+route (40GB/S bandwidth) 400 cycles/100 ns best-case DRAM latency. 10 cycle directory access latency

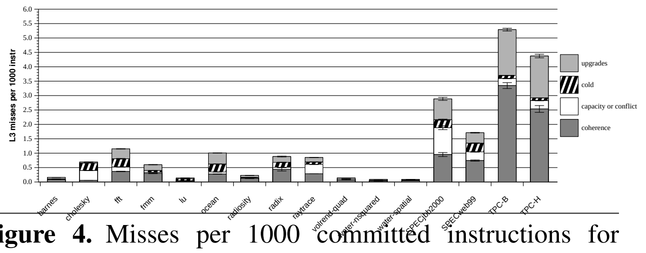

Figure 4. Misses per 1000 committed instructions for 16MB L3 cache.

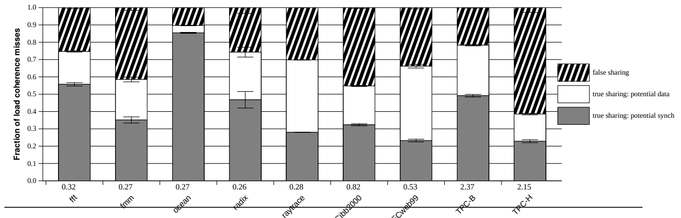

Figure 5. Breakdown of coherence misses caused by load instructions. (The number of load coherence misses per 1000 committed instructions labeled beneath each bar.)

transactions, caused by writes that touch a shared copy of the line, creating inter-processor communication but no data transfer. Many of the applications incur a significant number of coherence misses, especially the four commercial workloads at the right side of the figure. Such misses cause significant performance penalties, particularly in home-based protocols where they must typically make three network hops: from the requester to the home node, from the home node to the current owner, and back to the requester. Because some of the applications do not incur many coherence misses (barnes, cholesky, lu, radiosity, volrend, water-nsquared, and water-spatial), we omit these applications from the rest of our results. We do not expect ECDC to significantly improve their performance.

ECDC should benefit applications most by reducing the average latency of load instructions, because write buffers are able to hide most of the performance degradation caused by upgrade misses for these applications. Figure 5 further breaks down those coherence misses caused by load instructions into three categories: false sharing misses, true sharing misses that reference potential synchronization memory locations, and true sharing misses that reference potential false sharing memory locations. False sharing and true sharing misses are differentiated using the Dubois classification [12]. We separate true sharing misses into potential data misses and potential synchronization misses by labeling a miss as potential synchronization if the referenced cache line has been touched by a load-linked or store conditional instruction during the simulation, and all other misses are labeled as data. This classification is only approximate, because a memory location that is used once for synchronization may later be reallocated for a different purpose, but will still be considered synchronization using this classification. Consequently, the number of misses labeled synchronization may be overestimated, and should be read as a rough estimate.

We expect ECDC to offer performance improvement for those misses that are caused by false-sharing, and for some truly shared misses to data. ECDC should not offer any performance improvement by reducing misses to truly shared synchronization data (the black portion of each bar), because these misses are likely fetching the release of a lock variable. Although this class of misses is significant for each application, it represents no more than half of all load coherence misses for any applications other than fft and ocean. At 73%, the TPC-H decision support benchmark contains the largest percentage of misses caused by false sharing and true data sharing.

This data indicates that coherence misses occur frequently enough that their avoidance should yield some performance benefits, particularly in the commercial applications SPECjbb2000, SPECweb99, TPC-H and TPC-B.

4. ECDC Performance

In the previous section, we demonstrated that there is potential for the ECDC protocol should it be able to keep a line in the stale state long enough to capture its next reference. Our evaluation is broken into two parts, a microbenchmark evaluation and an evaluation using the applications characterized above.

Microbenchmark evaluation

As described in Section 2.1, ECDC offers performance improvement potential for parallel applications that share linked data structures. In this section, we compare the performance of ECDC to a conventional coherence protocol when running a lock-free list insertion microbenchmark, in the context of a 16-processor machine. We use Michael's hazard pointer-based lock-free parallel list maintenance algorithm for our microbenchmark's implementation [26].

The microbenchmark consists of a set of threads randomly inserting, deleting, or searching a linked list with some probability, where the probability x of a list insertion is always the same as the probability of a deletion. Each operation randomly chooses a node for which it will search, delete, or insert a new subsequent node. We use 15 threads

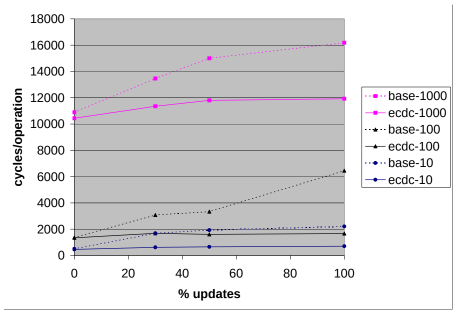

Figure 6. Lock-free list insertion microbenchmark performance.

running this mix of operations, and a single thread whose search operation latency is timed. Figure 6 charts the average list search latency, varying the x parameter from 0 to 50, resulting in the percentage of list modification operations ranging from 0% to 100%. The test was performed using three different average list lengths: 10, 100, and 1000, with larger list lengths decreasing the amount of contention in the microbenchmark.

As one would expect, as the fraction of update operations increases the average search time for the baseline machine also increases. For the highly contended list of length 10, the time per search increases by a factor of 4.2. As the fraction of update operations increases, the performance levels off; a point is reached with such a short list length at which contention is high enough that the probability of a cache miss occurring no longer increases. This is not true for the longer list lengths, where performance continues to degrade as the fraction of updates increases. For the list length of 100, the performance with 100% updates is 4.7 times worse than the performance with no updates. Moving to the 1000 entry list, performance is less affected by the updates because there is less contention, but degradation is still significant (48% worse performance) when moving from no updates to 100% updates.

When using the ECDC protocol, performance stays relatively flat as the percentage of updates is increased, because list searches are able to avoid many coherence misses while traversing the list. The performance is not completely flat, because some misses inevitably occur, creating a causal dependence on a recent write that forces many of the reader's stale lines to the invalid state. However, the ECDC protocol obtains significant speedups relative to the conventional protocol, measuring 2.74, 1.82, and 1.18 for the list of length 10, 100, and 1000 respectively, when 30% of the operations are updates. When 100% of the operations are updates, the ECDC protocol improves performance even more, with speedups of 3.11, 3.87, and 1.35 for these list lengths.

Application evaluation In this section, we evaluate the performance of three variations of the ECDC protocol relative to a conventional coherence protocol. In addition to the full-blown ECDC protocol (labeled ECDC base in each

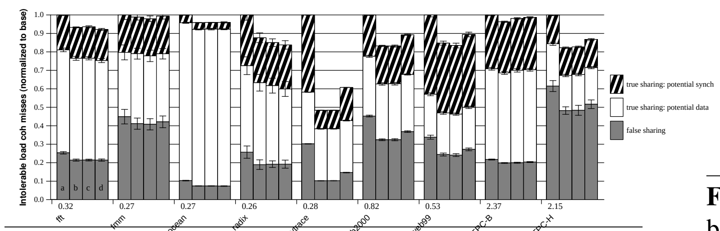

Figure 7. Reduction in load coherence misses. (a) baseline (b) Idealized ECDC (c) ECDC with merged r/w sets (d) ECDC with scalar probe sets. The number of load coherence misses per 1000 instructions is labeled beneath each bar.

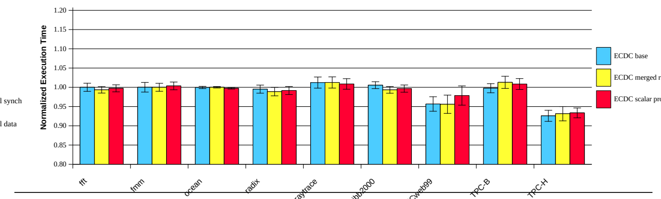

Figure 8. Performance: ECDC execution time relative to baseline.

chart), we also evaluate a variation in which we maintain a single probe set per memory location by using a single timer index table mapping in the PPB (labeled ECDC merged rw sets), rather than the two mappings that the base ECDC protocol uses to precisely implement the read and write upstream sets for each line. We also evaluate a variation of the protocol that uses a scalar timeout value to represent probe sets (labeled ECDC scalar probe set), rather than maintaining a vector of entries to individually track a processor's causal dependencies on every other processor.

A measure of the protocol's ability to use stale data is presented in Figure 7. We define an intolerable load miss as those load misses to lines in the invalid state. A reference to a stale line is tolerable because it returns stale data, but the reference may also initiate a coherence transaction (if it is determined to be a synchronization reference). Only part of this reduction in intolerable misses will result in improved performance, because for some of these misses the processor may simply continue to poll a memory location waiting for a new value to appear, without accomplishing any useful work. Each bar in Figure 7 is broken into true sharing misses and false sharing misses, and the true sharing component is further broken into potential synchronization and potential data misses. The rightmost three bars correspond to the bars in the prior figure (ECDC base, ECDC merged r/w sets, ECDC scalar), and the left-most bar corresponds to the baseline conventional machine.

We find that for many applications a significant fraction of intolerable misses is removed when using the ECDC protocol. In raytrace, the application with the largest reduction, as many as 52% of the intolerable misses are eliminated for the base ECDC protocol. However, approximately half of these misses are potential synchronization to truly shared data, which will probably not yield performance improvement. The other half of the reduction comes from false sharing misses. Across the remainder of the applications, nearly all of the reduction comes from these categories; there is very little reduction in misses to truly shared data. It is our understanding that AIX does not use any lock-free algorithms, and this set of applications does not include any convergent iterative algorithms, in which we would expect to observe a reduction in intolerable misses to truly shared data. From this data, it appears that any performance gains from the ECDC protocol will come from its reductions in false sharing miss, for which there is a significant amount for many of the applications.

In terms of the relative ability of each of the three ECDC implementations to avoid misses, the ECDC protocol with merged read and write sets performs similarly to the base ECDC protocol across all of the applications. The scalar ECDC implementation, although sacrificing some of the gains from the base ECDC implementation, still performs quite well despite the significant reduction in state from the vector-based representation.

Figure 8 presents the most important metric, the normalized execution time of each of the three ECDC variations relative to the baseline machine. Unfortunately, for most applications, the ECDC protocol has little effect on performance. There are two applications, SPECweb99 and TPC-H, in which the ECDC protocol offers measurable improvements in performance (4% and 8%, respectively, for the base ECDC implementation). Of all the applications, ECDC should improve TPC-H most, because TPC-H has a significant number of coherence misses, most of which are caused by false sharing, and of all the applications its coherence misses occur the most quickly after the line was invalidated, meaning that covering the miss with ECDC can be made practical.

SPECweb99 does not incur as many load coherence misses as TPC-H, so there is less potential for performance improvement. Although the average distance from invalidation to subsequent load coherence miss is much longer in SPECweb99 than in TPC-H, the ECDC protocol is able to retain cache lines for a longer period of time than in TPC-H (approximately 50,000 cycles as opposed to 5,000 cycles). Consequently, a significant fraction of false sharing misses are avoided.

For the scientific applications, coherence misses simply do not occur frequently enough for ECDC's small reduction in misses to create a significant performance benefit. With the exception of fmm, these applications are dominated by true-sharing misses. Although fmm contains a significant number of false sharing misses, the ECDC implementations are not able to eliminate these misses, indicating that before a processor is able to use a falsely shared stale data line, it usually becomes causally dependent upon a more recent operation by the processor that invalidated the line.

Of the commercial applications, the ECDC implementation is not able to improve the performance of either SPECjbb2000 or TPC-B. TPC-B is dominated by true sharing misses, and the false sharing misses that TPC-B does exhibit are not avoided by ECDC because lines are discarded soon after receiving invalidation messages (6600 cycles later, on average, for the base ECDC protocol) due to an incoming dependence on another line). Although ECDC is able to eliminate a significant fraction of false sharing misses in SPECjbb2000 (30% of all false sharing misses, representing 14% of all load coherence misses), SPECjbb2000 also incurs a significant number of non-coherence misses, watering down any performance gains from a reduction in coherence misses.

The slight performance degradation that occurs in a few applications (raytrace, SPECjbb, TPC-B) is due to the infinite probe timers used for this set of data, which results in some applications polling a memory location for an excessively long period of time before the processor finally becomes causally dependent upon that write, allowing the processor to observe the new value. This is a result of imperfect critical write/polling detection. When evaluating a variant of ECDC that expires cache lines after a fixed period of time, these slight degradations disappear.

5. Related Work

There has been a significant amount of related work on mechanisms that prevent the performance penalty associated with inter-thread communication, including optimizations at the algorithm level, language level, compiler level, and run-time system/hardware implementation level. The discussion here will be limited to those techniques that affect the shared-memory implementation, whether it be a hardware-based implementation or software-based implementation. We limit this discussion to other single-writer protocols that improve communication performance through the use of stale values or by delaying the observance of writes, and related work that specifically targets the false sharing problem.

5.1 Hardware systems

ECDC is closely related to prior implementations of delayed consistency. Motivated by the problem of false sharing, Dubois et al. proposed the first delayed consistency protocols, which delayed either the sending of all invalidates (sender-delayed protocols) or the application of all invalidates (receiver-delayed protocols) or both until a processor performs a synchronization operation [10][11]. Their work found significant reductions in cache miss rates from delayed consistency, however their studies did not determine if performance benefits could be obtained from these reductions. Dahlgren and Stenstrom more thoroughly explore sender delayed protocols implemented through the addition of a write cache that buffers outbound invalidate messages until an acquire or release is performed [8]. Their work focuses on update protocols and hybrid update/invalidate protocols. These proposals have demonstrated a reduction in multiprocessor coherence misses, but unfortunately each relies on properly-labeled synchronization operations. ECDC overcomes this obstacle, through heuristics that capture common synchronization constructs, and timeout mechanisms to prevent permanently delaying write observance for synchronization that is not captured by the heuristics. In addition, edge chasing delayed consistency extends the useful lifetime of stale cache lines by allowing a processor to use them until that processor becomes causally dependent on a newer copy of the cache line. In this sense, ECDC approximates the behavior of the entry consistency model, which orders operations that are related to one another (e.g., stores to the same data structure), while eliminating ordering requirements for operations that don't need to be ordered [3]. In comparison, the work by Dubois et al. and Dahlgren and Stenstrom take an all or nothing approach to the delaying of writes, in which all writes are delayed until a synchronization operation occurs, after which all pending invalidations are applied. The prior studies by Dubois et al. and Dahlgren and Stenstrom have also been limited in terms of experimental methodology. Demonstrating a reduction in miss rates is a positive outcome, however such reductions do not necessarily lead to performance improvement.

Lebeck and Wood's work on dynamic self-invalidation also included support for a form of receiver-delayed consistency by marking certain lines in a directory-based coherence protocol as "tear-off blocks", which would be automatically invalidated by the cache controller upon the next miss (under sequential consistency) or the next synchronizing operation (under weak ordering) [21]. This work focused on the reduction in write latency gained by shortening the list of sharers and thus avoiding the corresponding invalidation/acknowledgment latency; the extent of benefit obtained from lengthening the useful lifetime of cache blocks was not reported but would be interesting to study in future work.

Lepak evaluates a mechanism that speculatively returns stale data on a coherence miss, allowing the processor to continue executing dependent instructions until the cache miss returns and the speculation has been verified [22]. Huh et al. also describe a class of speculative protocols which include the mechanism proposed by Lepak [15]. Their mechanism increases the accuracy of stale data speculation by occasionally updating the stale data or by using a confidence predictor to decrease the number of misspeculations. These speculative protocols are complementary to the ECDC mechanism presented here if used by a processor that supports speculation. When the ECDC mechanism indicates that a stale cache line can be used, it can be used non-speculatively, allowing the processor to commit the instruction, whereas the speculative mechanism would have forced the instruction to wait for the cache miss to return, potentially stalling the machine. The speculative mechanism will be useful in cases where the ECDC protocol indicates that it is not safe to use stale data.

There have been several proposed hardware mechanisms that attack the false sharing problem in addition to the work described above. Dubnicki and Leblanc evaluate a coherence protocol that utilizes an adjustable line size, dynamically detecting false sharing misses and splitting cache lines in half accordingly [9]. Chen and Dubois propose a sub-blocked cache in only part of a cache line are invalidated, allowing other parts of the cache line to be used [7]. Anderson and Baer propose a similar protocol that uses a large transfer size and small coherence unit in order to take advantage of spatial locality while avoiding false sharing [2]. Each of these proposals is successful at reducing false sharing misses, and will be able to more successfully avoid false sharing than ECDC however their benefits will be limited to false sharing.

Afek et al. and Brown describe theoretical delayed consistency algorithms similar to Dubois et al.'s in the context of update-based coherence protocols and invalidate-based protocols, respectively [1, 4]. These algorithms limit a processor's use of stale values to those that are more recent than the most recent write by that processor. Their work includes formal presentations of the caching algorithms, but does not include any experimental evaluation.

Rechtschaffen and Ekanadham describe a heuristic-based cache coherence protocol that allows a processor to modify a cache line while other processors continue to use a stale copy [28]. Published only in patent form, their proposal is accompanied by no experimental data. Comparing ECDC to this and other prior art would make interesting future work.

5.2 Software systems

Keleher, Cox, and Zwaenepoel's lazy release consistency protocol is a sender delayed protocol that is similar to the receiver-delayed edge-chasing implementation discussed here [16]. Like ECDC, Keleher's proposal delays the observance of writes until a processor's read operation becomes causally dependent upon a write, at which point all writes that causally precede the observed write will become observable to the reading processor. Like the hardware-based delayed consistency work, lazy release consistency is dependent upon properly labeled synchronization operations, limiting its applicability to current architectures. Also, because lazy release consistency was proposed and evaluated in the context of a software-based distributed memory machine managing coherence granularity at the size of a page, the trade-offs involved are quite different from those involved in a hardware based multiprocessor. An interesting avenue of future work would be to compare the overheads of a software-based implementation of ECDC to lazy release consistency.

Tambat and Vajapayem present the performance advantages of a non-blocking memory access primitive called Global_Read in the context of a software-based distributed shared memory machine [29]. The Global_Read primitive returns new data once communication has completed, but until that time returns the previous copy of the data, allowing the application make forward progress using stale data until the new data is locally available. At a high-level, this mechanism is similar to the ECDC protocol presented here, because they both allow stale shared data to be read in order to tolerate communication latency. However, the Global_Read primitive provides an explicit interface to the programmer who can then dictate whether or not stale data should be used for a particular access. This is a powerful mechanism which can achieve similar benefits as ECDC when a programmer has written an application to use it.

McKenney and Slingwine describe Read-Copy Update, a software technique that creates multiple versions of a data structure allowing concurrent readers in an older version while a new version can be created by a writer [25]. When the writer has finished its update(s), it publishes the new version such that subsequent readers will operate on the new version. Wang and Weihl describe a software caching algorithm used in an implementation of concurrent B-trees that allows multiple versions of memory such that local stale versions of the B-tree can be read without incurring a cache miss, saving the latency of fetching the new version [30]. Their implementation improves performance over 300% for a highly contended B-tree microbenchmark. The ECDC protocol can provide similar performance benefits as multiversion memory and RCU in some circumstances, however the ECDC benefits may not be achievable if the data structure isn't cache resident, whereas the software techniques are independent of data size.

6. Conclusions

In our evaluation, we have shown that the performance of some applications can be improved by edge-chasing delayed consistency. Of four commercial workloads studied, ECDC improves the performance of two, TPC-H and SPECweb99, by 8% and 4% respectively. We find ECDC has little effect on the performance of the SPLASH2 scientific applications studied here.

While the idea of using stale data is enticing, our study shows that the benefits of such delay are small, and that the majority of these benefits from delayed invalidation come from mitigating the false sharing problem, rather than any tolerance of races or an application's ability to consume stale data in a productive manner. Since false sharing can be effectively solved a number of other ways, through either padding in software or less complicated hardware mechanisms, it is not clear that the advantages of the ECDC mechanism outweigh the hardware complexity and overheads.

Although our observations have been empirically evaluated in only a few applications, it is unclear to us why these results are not indicative of other applications using lock-based critical sections. For this reason, we believe that future work in this area should be application-driven; one should ask what would allow an application to productively use an old version of data, and only then optimize such behavior.

References

[1] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1): 182–205, January 1993.

[2] C. Anderson and J.-L. Baer. Design and evaluation of a subblock cache coherence protocol for bus-based multiprocessors. Technical Report UW-CSE-94-05-02, University of Washington, May 1994.

[3] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA, 1991.

[4] G. Brown. Asynchronous multicaches. *Distributed Computing*, 4(1):31–36, 1990.

[5] H. W. Cain. *Detecting and Exploiting Causal Relationships in Hardware Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin, 2004.

[6] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proc. of the Symp on Principles of Distributed Computing*, pages 157–164, August 1982.

[7] Y.-S. Chen and M. Dubois. Cache protocols with partial block invalidations. In *Proc. of the Seventh Intl. Parallel Processing Symp.*, pages 16–24, 1993.

[8] F. Dahlgren and P. Stenstrom. Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 26(2):193–210, April 1995.

[9] C. Dubnicki and T. J. LeBlanc. Adjustable block size coherent caches. In *Proc. of the 19th Intl. Symp. on Computer Architecture*, pages 170–180, 1992.

[10] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th Intl. Symp. on Computer Architecture*, pages 434–442, June 1986.

[11] M. Dubois, J.-C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Supercomputing*, pages 197–206, 1991.

[12] M. Dubois, J. Skeppstedt, and P. Stenstrom. Essential misses and data traffic in coherence protocols. *Journal of Parallel and Distributed Computing*, 29(2):108–125, 1995.

[13] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proc. of the 15th Intl. Symp. on Computer Architecture*, 1988.

[14] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, 1987.

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

[16] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Intl. Symp. on Computer Architecture*, 1992.

[17] E. Knapp. Deadlock detection in distributed databases. *Computing Surveys*, 19(4):303–328, 1987.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] A. Landin, E. Hagersten, and S. Haridi. Race-free interconnection networks and multiprocessor consistency. In *Proc. of the 18th Intl. Symp. on Comp. Architecture*, 1991.

[20] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, pages 241–251, 1997.

[21] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, pages 48–59, 1995. ISBN 0-89791-698-0.

[22] K. M. Lepak. *Exploring, Defining, and Exploiting Recent Store Value Locality*. PhD thesis, University of Wisconsin-Madison, 2003.

[23] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proc. of the 27th Intl. Symp. on Computer Architecture*, June 2000.

[24] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[25] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[26] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. of the 2004 Conf. on Programming Language Design and Implementation*, pages 35–46, 2004.

[27] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 network architecture. *IEEE Micro*, 22(1): 26–35, 2002.

[28] R. N. Rechtschaffen and K. Ekanadham. Multi-processor cache coherency protocol allowing asynchronous modification of cache data. United States Patent 5,787,477, July 1998.

[29] S. V. Tambat and S. Vajapeyam. Non-strict cache coherence: Exploiting data-race tolerance in emerging applications. In *Proc. of the 2000 Intl. Conf. on Parallel Processing*, August 2000.

[30] P. Wang and W. E. Weihl. Scalable concurrent b-trees using multi-version memory. *Journal of Parallel and Distributed Computing*, 32(1):28–48, 1996.

[31] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, pages 24–36, June 1995.