Reducing Coherence Overheads with Multi-line Invalidation (MLI) Messages

Hongil Yoon and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI, USA
{ongal, sohi}@cs.wisc.edu

Abstract

Most multiprocessors employ coherent caches despite the overheads of doing so. As future processors will be multiprocessors with elaborate cache hierarchies, the overheads of cache coherence will be an important area for efficiency improvements. This paper proposes a novel technique, called Multi-Line Invalidation (MLI) messages, to reduce an important aspect of coherence overhead—the invalidation traffic—by combining multiple traditional (single-line) invalidation messages into a single message. MLI messages can be used alongside any traditional coherence protocols. Two empirical phenomena—the coarse-grain nature of data sharing and the trend towards programs that are free of data races—enhance the utility of MLI messages.

This paper illustrates how MLI messages could be constructed and deployed alongside an existing coherence protocol. It then presents an evaluation of their effectiveness in reducing the number of invalidation messages for several benchmark programs. We find that in several cases a significant reduction in the overall address network traffic and energy consumption could be achieved without performance degradation.

1. Introduction

Coherent caches have long been considered to be one of the most successful innovations in computer architecture, allowing low-latency access to data in shared memory multiprocessors without explicit software management. Coherence is maintained with a coherence protocol that typically uses invalidation messages to invalidate copies of cache lines that are stale. These invalidation messages increase network traffic. They also consume energy. Today, despite the complexity and overheads of doing so, coherent caches are ubiquitous.

Over the years there has been a plethora of work proposing optimizations to improve the effectiveness of coherence protocols, reduce their overheads, reduce their implementation complexity, and improve their energy efficiency. Recently, multiprocessors have become ubiquitous, with not only an increasing number of cores but also with non-traditional architectures such as heterogeneous systems. To facilitate the ease of programming such systems, coherent caches will be important [17][30][33]. Accordingly, the issue of reducing cache coherence overheads, especially invalidation traffic, will be of increasing importance.

This paper proposes a novel approach to significantly reduce the invalidation traffic while continuing to use a workhorse, well-understood, and time-tested coherence protocol. It is based upon a combination of two known observations whose empirical importance is likely to increase in a parallelism heavy future: (i) if race conditions do not exist invalidation messages can be delayed; they do not have to be performed in a demand fashion, and (ii) due to the coarse-grain nature of sharing, invalidation messages to proximate lines in *a larger region of memory* (e.g., a page) occur in temporal proximity. Thus, by delaying an invalidation message to a cache line, it could be combined with invalidation messages to other cache lines in the same region, and the invalidations of multiple proximate cache lines could be conveyed in a single, multi-line invalidation message.

In this paper, we make three main contributions:

- First, we propose a simple microarchitectural technique, called *Multi-Line Invalidation (MLI) messages*, to reduce invalidation traffic by leveraging two widely known concepts: the coarse-grain data sharing and delaying of invalidation requests.
- Second, we demonstrate how our proposed technique can be implemented alongside a conventional coherence protocol and a widely-deployed memory consistency model (Total Store Order (TSO)), *without modifications to the workhorse components* of the coherence protocol and without any requirements of software.
- Third, using a full system simulator, we evaluate MLI messages, not only for a standard homogeneous multiprocessor, but also for a heterogeneous processor (e.g., CPUs with a GPU). The results demonstrate a significant reduction in invalidation traffic as well as in address network energy consumption, along with some performance benefits, in several cases.

The paper starts out by discussing the previously known concepts of exploiting coarse-grain sharing and the delaying of invalidation requests, whose combination leads to the main idea of the paper (section 2). Then it presents details of how messages that invalidate multiple lines are created dynamically and used, and the issues that must be dealt with (section 3). Then section 4 presents the results of an experimental evaluation assessing the effectiveness of multi-line invalidation messages. Additional related work is discussed in section 5 before concluding in section 6.

2. Background, Select Prior Work and Main Idea

Cache-based systems have three primary sources of message traffic: (i) Read misses, (ii) Write misses, and (iii) Invalidations. The first two occur in any cache-based system, coherent or not; the third is considered an overhead of maintaining coherent caches [26]. Over the past three decades, there has been a plethora of work on improving multiple aspects of cache operation. In this paper we are concerned with invalidation traffic, as that is the main overhead of maintaining coherence. In order to set up the main idea, we start out by discussing two threads of work pertaining to invalidation traffic that are directly related to our main idea: exploiting the coarse-grain nature of data sharing and the timing of coherence operations.

2.1 Coarse-Grain Coherence Techniques

Normally cache coherence is carried out at a cache line granularity. However, programs employ data structures that generally are much larger than cache lines and typically occupy multiple contiguous cache lines. Moreover, data sharing is a property of how a program modifies and accesses its data structures; how the components of a data structure are shared determines the sharing patterns of cache lines that contain them. Thus we can expect multiple cache lines—the ones containing the different components of the data structure—to have similar (or the same) access and sharing patterns and thus requiring similar coherence operations. This phenomenon underlies a recent body of work on coarse-grain coherence tracking techniques.

Moshovos, *et al.,* propose Jetty as a means to track a superset the blocks being cached (or not cached) and use this information to reduce the number of cache tag lookups [28]. Another proposal to exploit more coarse-grain access behavior is the Page Sharing Table (PST) of Eckman, *et al*. [15], where each cache maintains precise information about the pages that it is caching; snoop requests are not submitted to the local cache if the PST indicates that the corresponding page is not present. These proposals work because multiple lines from the same page are likely to reside in a cache.

Other proposals exploit the coarse-grain sharing phenomenon to reduce the number of broadcast invalidation requests. Here, in addition to a normal coherence protocol, each processor contains additional hardware for monitoring the coherence status of larger regions of memory. When an invalidation request for a cache line is received, the response includes the local coherence status of the entire region, not only the cache line. If these responses from remote caches indicate that they are not caching other lines from the region, the requesting processor can freely write to lines in the region without submitting further invalidation requests. Moshovos proposes RegionScout [27] and Cantin, *et al.,* propose Region Coherence Arrays [7] to reduce the number of broadcasts of invalidation requests.

In a similar vein, Wallin, *et al.*, propose a bundling technique to reduce the prefetching overhead for snoop-based systems. Multiple prefetch read requests are bundled, using a prefetch bit mask to identify the requested lines, and the bundled prefetch request combined with a coherence request [36]. They also propose a bundling technique for reducing the overhead of upgrade requests. This requires a (slight) modification to the snooping coherence protocol.

2.2 Timing of Operations

Of the three request types emanating from a cache: (i) a Read miss, (ii) a Write miss, and (iii) a coherence (invalidation) operation, the first two are normally handled in a demand fashion, i.e., are processed as soon as possible, since the processor can't proceed too much further with correct operation until the missing line is brought into the cache. The third type of request, which is the overhead of cache coherence, does not return a line. Rather the request's completion delivers the necessary permissions for correct (coherent) operations to the cache line. Canonically an invalidation was also handled in a demand fashion, though delaying its timing can lead to many optimizations.

Early work by Scheurich observed that an invalidation can be delayed until the subsequent cache miss without any concern while still maintaining sequential consistency [32]. If programmers employ some discipline in writing their programs, and can thus avail of weaker memory models, even more opportunities for delaying invalidations arise. Dubois, *et al*., proposed the concept of delayed consistency where invalidation operations to non-synchronization variables can be delayed until the occurrence of the next synchronization variable access, allowing significant other activity to be overlapped with an invalidation request [14]. Similarly, release consistency allows invalidation operations to be delayed until the release of a synchronization variable [1]. Dahlgren, *et al.*, proposed delaying write updates to different words in a cache line between synchronizations, under relaxed consistency models, for write-update coherence protocols [12], with the updated words being tracked with a bit mask, thereby lowering network traffic in a write-update protocol. They also applied the same scheme to reducing the overhead from false-sharing misses for invalidate protocols. Keleher, *et al.,* further relax the timing constraints in their proposal of lazy release consistency for a distributed shared memory (DSM) system, where invalidations have until the next acquire operation to complete [24].

2.3 Lessons from Related Work and the Main Idea

A lesson from the effectiveness of coarse-grain coherence optimization techniques is that cache access and coherence operations tend to occur on spatially-proximate cache lines in a temporally-proximate manner. A lesson from the plethora of work on the timing of coherence operations is that many invalidation operations don't have to be carried out hurriedly; they could be delayed for quite a while without resulting in erroneous operation, especially if the program is free of data races.

We get to the main idea of the paper by combining the above two lessons: *if we delay invalidation requests sufficiently so that invalidation requests to multiple different*

lines from a larger region of memory (e.g., a page) are pending, the multiple requests can be combined into a single (slightly more complex) Multi-Line Invalidation (MLI) message. Using an MLI message—which is then broken down into multiple single-line invalidation operations at a remote directory and/or cache—instead of multiple single-line invalidation messages, can significantly reduce the number of invalidation messages sent, thus significantly lowering the message overheads of maintaining cache coherence. Importantly, *this can be done without any changes to the core operations of the traditional coherence protocols.*

3. Coherence with Multi-Line Invalidation Messages

3.1 Overview

The coherence hardware in a multiprocessor can be viewed as a collection of interacting finite-state machines (FSMs). These FSMs, which are a part of cache controllers and directory controllers, carry out actions in response to messages they receive, using and updating state that they maintain locally. They also send messages to other FSMs.

Invalidation messages communicated between the FSMs concern a single cache line. The recipient of an invalidation message does not care how or when the message arrived; it simply performs the necessary actions and provides a response. In particular, it does not care whether an invalidation for a line arrived in isolation, or in close temporal proximity to invalidation messages to other cache lines, or even combined with other invalidation messages.

We propose to *make no changes to the actions of the FSMs of a coherence protocol*. Such FSMs are notoriously difficult to get correct, and once designed correctly, designers are very reluctant to start from scratch on a new FSM. Rather, we propose to *change the messages that are used to communicate actions* between the different FSMs of the overall coherence protocol: when possible, multiple messages are aggregated into a single multi-line message by a sender, which is then decomposed into the corresponding multiple, single-line messages by a receiver. Otherwise, normal single-line messaging is used.

For messages to be combined with other messages, they have to be delayed so that they can meet up with other messages. Messages can only be delayed when it is safe to do so. For ease of explanation, let us suppose that regions of program execution where it is safe to delay invalidation messages can be identified. We call this an *MLI Region*; markers *MLI_Begin* and *MLI_End* indicate the beginning and the end of such a region, respectively.

When in an MLI Region, an *MLI Unit* with a cache controller tries to merge multiple invalidation messages for the same region of memory into MLI messages. When it is no longer safe to continue delaying invalidations, all pending invalidation messages being held up by a cache controller are sent out. We will see the basic MLI messages and how they are created and used further in section 3.2.

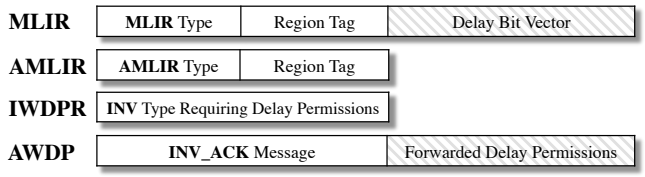


Figure 1. Multi-line invalidation message types

For cache lines that may contain falsely shared data, delaying invalidations can create additional work: different words of the line may have been updated locally and the updates will need to be merged to construct the correct contents of the cache line. To avoid this merging, we want to prevent delaying invalidations, and use normal invalidation operations, to lines that may be falsely shared. We do so with the concept of *delay permissions* that we describe in section 3.3.

When invalidation operations can be delayed, for how long they can continue to be delayed, and by when they have to complete for correct operation, is dictated by the memory consistency model. That is, the implementation of the memory consistency model is responsible for determining the *MLI_Begin* and *MLI_End* markers. Our objective is to utilize MLI messages with widely deployed coherence protocols and consistency models, so that they are completely transparent. In particular, we consider the *Total Store Order (TSO)* memory consistency model, and show how MLI messages could be deployed, and how the markers are generated, while maintaining TSO, completely in the microarchitecture, without any programmer/software involvement, in section 3.5.

3.2 Basic Operation

We describe the proposal alongside a directory-based coherence scheme since we believe directory schemes will be dominantly going forward. However, the concepts are equally applicable with broadcast-invalidate schemes. Moreover, we initially assume a single-bank LLC/memory where the directory resides; we will discuss what happens for a multi-bank, interleaved, directory in section 3.6.

In addition to normal single-line coherence messages, we propose new message types that contain aggregate information for multiple lines, along with other supporting messages. These are illustrated in Figure 1.

The main additional message type is *a multi-line invalidation request (MLIR) message*. This message is comprised of the address of a larger region of memory (e.g., a page), and a bit vector indicating the identities of the (multiple) lines in that region that are to be invalidated.

An MLIR message is sent to the directory of the home node(s) of the corresponding memory region. In turn, the directory controller converts it into (zero or more) MLIR messages that are forwarded to the relevant remote caches. At a remote cache an MLIR message is broken down into the corresponding (multiple) single-line invalidation messages which are submitted to the local coherence FSM, which responds to them as normal. *Ack_MLIR (AMLIR*)

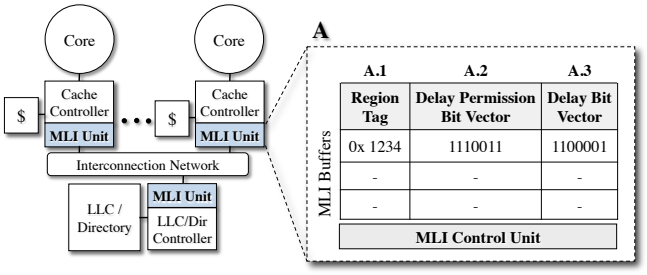


Figure 2. Support for multi-line invalidation messages

invalidation acknowledgment messages, if needed, are returned from the directory/cache to the requestor after all the requisite invalidation operations are completed. We will discuss the other 2 message types in Figure 1 in the next section.

Figure 2 shows an overview of the scheme. Alongside a normal cache-coherent multiprocessor we have added some additional hardware (*MLI Unit*s) at the caches/directories to construct and respond to the new message types.

An MLI unit at a cache controller tries to create MLIR messages. To do so it has *merging hardware* (Box A in Figure 2) whose main component is a set of buffers (MLI buffers) and associated logic. Each buffer is tagged (A.1) with the identity of the corresponding region of memory (e.g., a page number) and contains a *delayed bit vector* (DBV (A.3)) for the lines to be invalidated in the region.

The default is normal coherence operation, using normal coherence protocol messaging and operations. At certain points in the execution of the program, MLI units are activated or deactivated in response to *MLI_Begin* and *MLI_End* events, respectively. We shall see how these events are created in section 3.5.

When activated, an MLI unit tries to delay invalidations and to create MLIR messages. It intercepts single-line invalidation requests emanating from the coherence FSM and delays them, if possible. It checks if there is an active buffer for the accessed region and, if so, the corresponding bit in the DBV is set to indicate a pending invalidation to that line. If there isn't a matching buffer, one is allocated, possibly evicting an active buffer.

When a buffer is to be evicted, an *MLIR* message, as per the corresponding DBV, is sent out, and the eviction completed when the corresponding *AMLIR* is received. When the line whose invalidation is being delayed is replaced, the corresponding buffer is evicted before writing back the cache line.

At an *MLI_End* event, all MLI buffers are evicted and corresponding MLIR messages sent, i.e., all pending invalidation operations are completed. Normal coherence protocol messaging is restored until the next *MLI_Begin* event.

3.3 Delay Permissions and False Sharing

False sharing occurs when different data being accessed by different processors are mapped to the same cache line [34]. It can be mitigated somewhat, or even eliminated, by proper variable placement [23]. However, in practice a programmer may not have complete control over their operating environment as code written by others may be used during the execution of their application program. Examples include libraries as well as the operating system.

False sharing can create complications when invalidations are being delayed [1][24][32]: different caches could have different versions of the lines and a process to combine the different versions would be required. This is a tedious process that could add unnecessary complication to the base coherence protocol. Accordingly what we desire is to delay invalidates only in cases where they would not result in *any* problems, and use normal coherence messaging otherwise.

To permit selective delaying of invalidations, we associate a *delay permission* with each line. The delay permission for a line is moved between a cache and the directory as needed; at most one cache can have a line's delay permission. The directory MLI unit maintains state indicating whether it has the delay permission for the line, or whether that permission has been given to a cache. A cache MLI unit maintains the set of delay permissions for the lines in an active region in a *permissions bit vector* (PBV (A.2) in Figure 2), and delays invalidations only to lines for which it has the delay permission; else it passes through the invalidations.

Initially the directory owns the delay permission for a line. Delay permissions for lines within a region are requested by a cache MLI unit with an *Invalidation With Delay Permissions Request (IWDPR)* message. This message is generated when the first invalidation to a new region is encountered by the MLI unit, or when the invalidation is to a line in a matching region for which there is no delay permission. In addition to the normal, single-line invalidation operation, it also serves as a request for delay permissions to other lines in the region.

The *Ack With Delay Permissions (AWDP)* message returned in response to an IWDPR message includes delay permissions to some lines in the region (determined by some algorithm), *excluding* the line for which the invalidation was initiated. This is because the IWDPR performs the necessary invalidations for that line and no further invalidations to that line will emanate without other intervening operations. This also permits a handling of false sharing as we shall see below. Returned permissions are used to update the corresponding PBV.

When a buffer is evicted from a cache MLI unit, the delay permissions held in its PBV are returned to the directory as part of the corresponding MLIR message.

False sharing is detected and handled as follows: when an INV (normal invalidation) or IWDPR request from the network is received by a cache MLI unit to a line for which it has a delayed invalidate, it is likely due to false sharing. (It can't be true sharing since it has been determined that the first write is in a race-free region, so other writes to the same address couldn't be happening.) When false sharing is detected, the delay permission for the line is returned to the directory, which then does not pass it to another cache. Consequently, for lines deemed to be potentially false

shared, further invalidates aren't delayed and normal coherence protocol messaging is employed.

3.4 Overall Operation

Table 1 and 2 below summarize the overall operation of the cache and directory MLI units, respectively, i.e., how the MLIR, AMLIR, IWDPR, AWDP messages are constructed and used. As in most research papers that propose interacting FSMs, we present only the high-level operations and low-level details have been omitted. It is important to note that the MLI units are simply observing normal operations of the coherence mechanism and merging multiple emanating actions to reduce the messages/traffic required. The operations of the base coherence protocol are not altered in any way; *the MLI units simply alter the timing of different coherence actions, and the means for delivering the required actions between the different components of the base coherence mechanism*. We discuss select important actions below; the others are self-explanatory.

Most coherence protocols employ the use of acknowledgment messages which indicate that an invalidation operation has completed. In some textbook protocols, these messages are sent from remote caches to the directory from where they are forwarded to the requesting cache. In most commercially-deployed protocols, the directory returns an acknowledgment message to the requesting cache indicating how many additional acknowledgment messages to expect and these messages are sent directly from the remote caches to the requester, which keeps a count of how many messages are still pending. The tables below assume the latter approach.

The top part of Table 1 summarizes how a cache MLI unit handles invalidation requests emanating from the processor (the local cache FSM). The bottom part describes how it responds to messages from the network side. These include the normal invalidate (INV) and acknowledgment (ACK) messages, as well as our proposed new message types.

When a cache MLI unit receives an INV, IWDPR, or MLIR message from the network, invalidates are sent to the corresponding cache line(s) (which may necessitate a write back of a cache line if it has been modified, as per a normal coherence protocol), and delay permissions for those lines (if present) are taken away from the cache MLI unit and returned to the directory MLI unit. AMLIR and ACK messages are used to track completion of the invalidation operation (just as in a normal protocol). For an AWDP message, the returned delay permissions are extracted and used to update the PBV.

At a directory MLI unit, an IWDPR is forwarded to the relevant remote caches and an AWDP (along with relevant delay permissions) returned to the requesting cache, which then awaits the necessary AWDP messages from the remote caches. For an MLIR request, in addition to forwarding the request to remote caches and returning an AMLI message to the requestor, the directory unit also updates its local set of owned delay permissions with the returned permissions.

Cache MLI Unit: Processor-Side Requests	

Invalidation from Processor	***If*** *matching MLI buffer*
	If delay permission available;
	Set corresponding DBV[i] bit;
	Send Ack to cache.
	Else
	Send IWDPR to Directory
	Else
	Evict LRU buffer;
	Allocate buffer for region;
	Send IWDPR to Directory

Cache MLI Unit: Network-Side Messages	
INV/IWDPR/ MLIR	Send relevant Invalidate(s) to cache; Release Delay Permission(s); Reset corresponding DBV[i] bit(s); **If** IWDPR Return AWDP. **If** MLIR return AMLIR.
AMLIR/ACK	***If*** last Ack of MLIR/INV Complete MLIR/INV request; ***Else*** Update Ack count.
AWDP	Update delay permissions; Forward Ack to Cache.

Table 1. Operation of a Cache MLI Unit

Directory MLI Unit: Network Messages	
IWDPR	Send IWDPR to remote caches Return AWDP to requestor
MLIR	Consult directory and send MLIR(s) to remote caches (if needed) Restore delay permissions Return AMLI

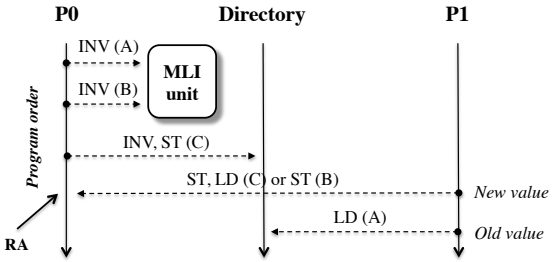
Table 2. Operation of a Directory MLI Unit

It is important to reiterate that the MLI units simply alter the timing of the individual actions of the base coherence protocol, but not the overall set of actions. For a given cache line, the FSMs of the base coherence protocol have the same actions, send the same requests for actions to, and receive the same requests from other FSMs, as they did without the presence of MLI units. The MLI units simply combine multiple requests for actions (invalidations) to different, proximate, cache lines into a single (albeit bigger) message over the communication network, rather than using multiple messages to convey the relevant actions.

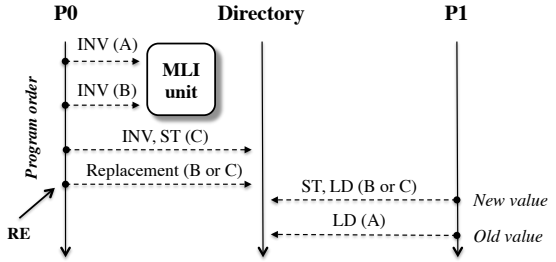
3.5 MLI Regions and the Memory Consistency Model

What is remaining is how the *MLI_Begin* and *MLI_End* events are generated. Recall that *MLI_Begin* is an indicator that it is considered safe to delay invalidation operations and *MLI_End* is an indicator that it is no longer safe to continue delaying them, i.e., all prior invalidation operations must be completed before execution proceeds past an *MLI_End*.

Furthermore, the ordering of the write operations is not maintained in an MLI region and there is no way of tracking the order in which the component invalidates of an MLIR message occurred. To sort this out, we turn to the memory consistency model which defines the consequences of the order (or lack thereof) of the store (invalidation) operations.



(a) Remote Access



(b) Replacement

Figure 3. Examples of store-store order violation with MLI messages

With a weak memory model, such as Release Consistency (RC), software can be given the responsibility of identifying race-free regions, which then become MLI regions. Software takes on the responsibility of inserting the *MLI_Begin* and *MLI_End* event markers. However, this approach is not consistent with our goals of applying MLI transparently for widely-deployed coherence protocols and memory models. We want the *MLI_Begin* and *MLI_End* events to be generated dynamically and transparently.

To understand how this can be accomplished, we observe that the MLI units at cache controllers can be viewed as (possibly) huge, non-FIFO, coalescing write buffers. We turn to the *Total Store Order (TSO)* memory model, which was proposed to accommodate FIFO *processor write buffers*. TSO is currently used in a variety of widely-deployed systems (including x86). To implement TSO correctly, the FIFO processor write buffers have to be drained (i.e., all pending operations completed) when a *buffer-draining* event occurs. Example events include memory fences, serializing instruction, interrupts, and others [19].

It is known that with coalescing (non-FIFO) *processor write buffers*, TSO can't be guaranteed. However, there is a big difference between processor write buffers and MLI units. To understand this, first observe that the local cache is the point where the local operations of a processor can be observed by another processor. Next, observe that a processor write buffer is between the processor and the cache whereas the MLI unit is between the cache and the network. This distinction plays an important role in permitting correct TSO operation, even without maintaining ordering of invalidations in the MLI units.

As mentioned earlier, the only changes we propose are in the messaging between a cache and other entities. In particular, no changes are made to the processor write buffers (which continue to be FIFO for TSO) or the coherence protocol. We are only concerned with what has to be done in the MLI units. Here observe that *by time an invalidation is registered with an MLI unit, the local cache has already been updated with the corresponding store data*. That is, the stores corresponding to the invalidations in the MLI unit have been performed, in FIFO order, at the point where they can be observed externally. The MLI units lose the FIFO order of the invalidations, but not the order of the stores. What is remaining then is to determine the situations where TSO could potentially be violated if the invalidations continue to be delayed, and address these situations.

To permit correct TSO operation, each event for which the processor write buffer needs to be flushed also becomes an *MLI_End* event. In addition, we need to make sure that when a processor P1 observes another processor P0's store, it must appear that all prior stores have been *completed*, i.e., all of them have been reflected in the overall memory order.

P1 can observe the results of P0's stores either if P1 makes a request to a line held in P0's cache, or to a line that used to reside in P0's cache but has was evicted and now resides elsewhere (from where P1 accesses it). For these cases, in Figure 3 we see the problems that can arise, and see how to handle them to ensure TSO.

Figure 3(a) illustrates the potential problem when P1 makes a remote request to data in P0's cache. Here line A is in a shared state in P0 and P1, and line B is in shared state in P0. P0 then writes to lines A and B, and the invalidations are delayed. These are followed by a store to line C, whose invalidation is not delayed and is sent out, resulting in line C in a modified state in P0. Later P1 makes a load or store request to line C, and this request ends up at P0's cache. If this requested is serviced normally, P1 could follow it would a read of line A, which would hit in P1's cache, and thus TSO violation would occur. A similar situation would result if P1 made a store request to line B, followed by a load to line A. (A load to line B followed by a load to line A would not violate TSO.) To prevent a TSO violation, an *MLI_End* event must be triggered at point RA, and all MLI buffers flushed, when: (i) there is a remote request to a line in modified state, or (ii) a remote write to a line for which an invalidation is being delayed. The *MLI_End* event must complete before responding to the remote request.

Figure 3(b) illustrates potential problems when a line is replaced. With the same initial state and sequence of events as in 3(a), suppose line B (or C) are replaced. P1 could then read or write line B (or C), and P0 would not be aware of its occurrence. A later (local) read of line A by P1 would violate TSO. To handle this case, an MLI_End event must be triggered at point RE, before a line in modified state, or a line to which an invalidation is being delayed, is replaced.

By triggering *MLI_End* events at necessary points, MLI messages could be employed *and be TSO compliant*.

MLI_Begin events can be triggered at the start of program execution and then at any point after an *MLI_End*.

3.6 Multi-bank Directories

We now consider the options with multi-bank directories. Here the lines of a contiguous region of memory could be spread across multiple banks, and thus the potential benefits of MLI messages could be reduced.

We consider two extremes: *region-level* and *line-level* interleaving. In the former case, all lines of a region are in a single bank, and thus the situation vis a vis traffic reduction is the same as a single-bank cache. In the latter case, we can consider the bit vector of an MLIR message to correspond to consecutive lines of the region mapped to the same bank. For example, with 2 banks the bits in an MLIR message would correspond to even (odd) lines if sent to an even (odd) bank.

We can also combine invalidations from consecutive, fixed-sized regions, using a fixed-sized bit-vector, as illustrated in Figure 4. Here we have 2 consecutive regions of 4 lines each, A (lines 0, 1, 2, 3) and B (lines 4, 5, 6, 7), and 2 banks with line-level interleaving. If we have a situation where lines 0, 1, 5, 6, and 7 need to be invalidated, we could do it: (i) with 4 messages, to bank 0 (line 0), bank 1 (line 1), bank 0 (line 6), and bank 1 (lines 5, 7), or (ii) with 2 messages, to bank 0 (lines 0, 6), and bank 1 (lines 1, 5, 7).

The multi-region combining approach above can also be used to construct normal MLI messages of variable length. We don't consider this further in this paper.

3.7 Prediction to Optimize Overheads

The MLI approach described above, while effective in reducing traffic when invalidations are to proximate lines, can actually result in worse traffic when invalidations can't be accumulated. Here is why: an IWDPR message invalidates its target line and returns a set of delay permissions to the local MLI unit. Permissions need to be returned to the directory when the corresponding MLI buffer is evicted. If no other lines in the region are invalidated, the MLIR message which returns these permissions is simply an additional overhead that does not perform useful work (invalidations). Likewise if the invalidation payload isn't sufficient, the additional bits of an MLIR message are overhead.

To alleviate these overheads, we can dynamically decide whether or not to use MLI for a particular memory region. To do so, we can employ a predictor that is indexed with the PC of a store instruction that triggered an IWDPR message which brings in the region, and which tracks the payload of the MLIR message when the corresponding region is evicted, to decide whether an IWDPR message or a normal INV message should be used.

4. Evaluation

In this section we evaluate the effectiveness of our proposal. We start out by describing the evaluation methodology and benchmarks (section 4.1). Our results are for a directory-based (MESI and MSI) coherence protocol. In sections 4.2, and 4.3 we consider the effectiveness of MLI messages in reducing the invalidation, and analyze characteristics of the MLI messages, respectively. In section 4.4, we analyze the reduction in the energy consumption on the address network from the reduced invalidation traffic. We present the results on a timing simulation in section 4.5.

4.1 Evaluation Methodology and Benchmarks

We evaluate two types of cache-coherent systems: (i) a classical *homogeneous* multiprocessor system consisting of multiple homogeneous processors, and (ii) a *heterogeneous* system, which has CPUs and GPU units, with the heterogeneous units having coherent caches. For the bulk of our experiments on a homogeneous system, we use the gem5 cycle-accurate, full-system X86 simulator with Garnet [2][5] and extend it to support MLI messages. Table 3 lists the base parameters of the simulated system. For evaluating MLI messages on a heterogeneous system with coherent caches, we extend gem5-gpu [16]. We have 7 parallel benchmarks for a multiprocessor (barneshut, ocean, pbzip2, fluidanimate, canneal, bodytrack, and lu) from [4][39], and we have 5 benchmarks for a heterogeneous system (streamcluster, backprop, bfs, kmeans, and lavaMD) from [8]. They are run to completion for the input. *MLI_Begin* and *MLI_End* events are dynamically generated without any software assistance of any kind.

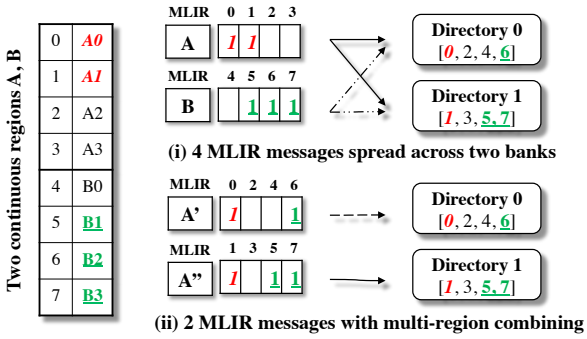


Figure 4. MLI messages and multi-bank directories

Processors	2 GHz, Timing Simple CPU (a single issue, in order)
Private Cache	Split 2 way I$ and D$ (varying sizes) 64Bytes cache line
Shared Cache	Varying sizes (Inclusive) 4 way-unified cache
Memory	4GB
Coherence	MSI and MESI directory-based protocol with exact sharing information

Table 3. Simulation model parameters

4.2 Reduction in Invalidation Traffic

Our first set of results concerns the reduction in total invalidation traffic on the address network. We concern ourselves with only the address network since our proposal does not result in any changes in the activity on the data network.

Our initial set of results is for the homogeneous system, as this allows us to get an understanding of the initial impact and characteristics of MLI messages. Figure 5 presents relative invalidation traffic (bytes transferred) for a system supporting MLI messages compared to the baseline, for 8

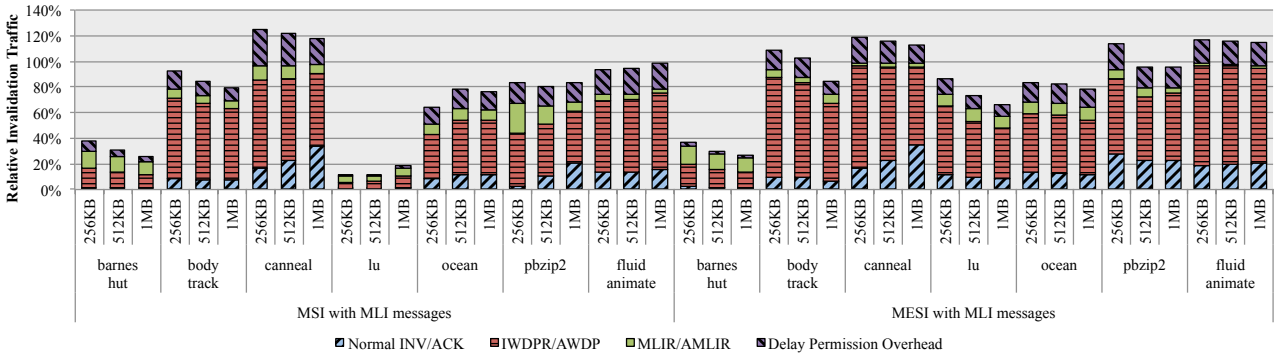


Figure 5. Relative Invalidation Traffic with MLI Messages (without any predictions)

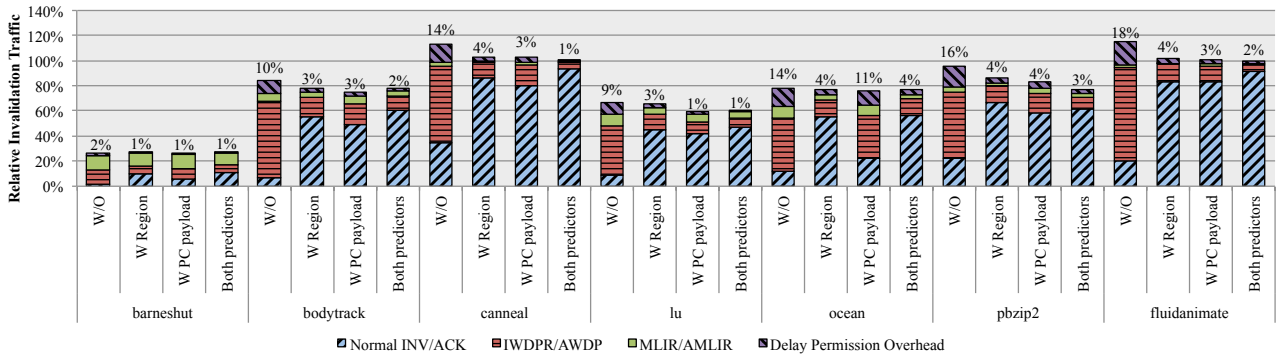
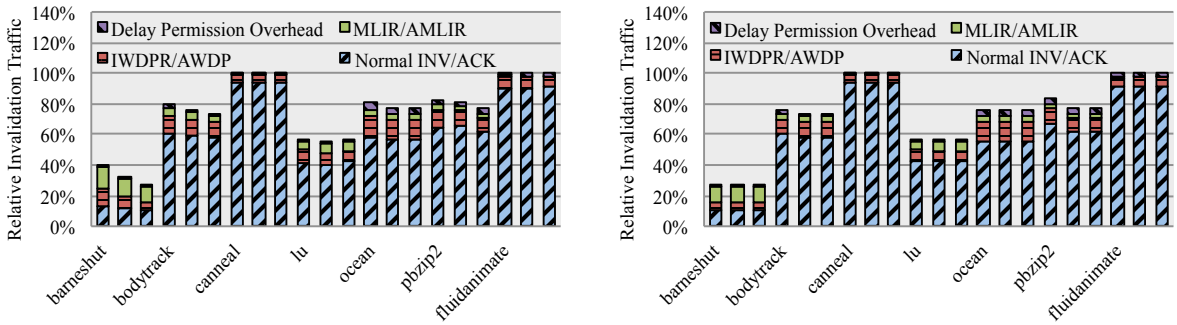


Figure 6. Impact of Varying Prediction Techniques (MESI with MLI messages)



(a) various region sizes (1KB, 2KB, and 4KB)

(b) various number of buffers (4, 16, and 32)

Figure 7. Impact of varying region sizes and number of MLI buffers

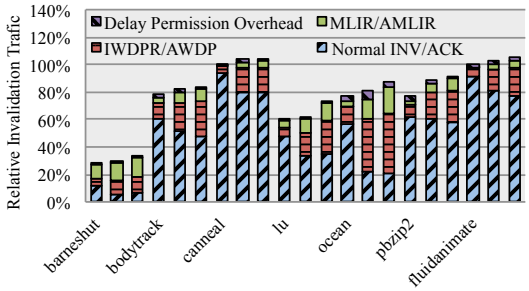


Figure 8. Impact of varying directory banks (1, 2, and 4)

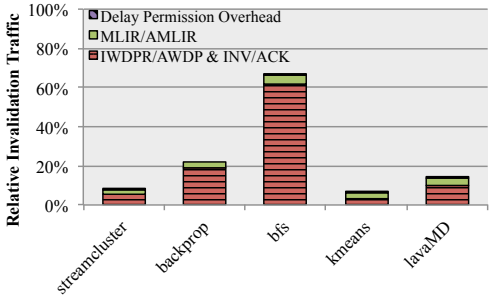


Figure 9. Relative Invalidation Traffic for a Heterogeneous Architecture with MLI Messages

cores, for both MSI and MESI coherence protocols. The left of the figure shows the results for MSI with MLI messages while the right is for MESI with MLI messages. For each benchmark, there are 3 bars, corresponding to private data cache sizes of 256KB, 512KB and 1MB. The number of Invalidations (MLIR/AMLI) in MLI regions, the middle component corresponds to other invalidations (IWDPR/AWDPR), and the bottom corresponds to normal invalidations (INV/ACK). The difference between the height of each bar and 100% corresponds to the reduction in invalidation traffic; we assume that MLIR messages are twice the size of normal invalidation messages (an additional 8 bytes to accommodate the bit vector for sixty four 64-bytes lines in a 4 KB region).

The results of these figures show significant benefits in reducing the number of bytes transferred, in many cases. For example, when running barneshut, about 70% of the invalidation traffic is reduced for the 512KB cache regardless of the coherence protocols. We do slightly worse in some cases, due to the overhead of getting delay permissions that are used (top part of each bar).

As mentioned in section 3.7, the above overhead can be reduced with a suitable predictor that predicts whether or not to deploy MLI, and we consider this next. Two distinct phenomena underlie the predictors: (i) MLI is effective in some regions of the program and not in others, (ii) lines invalidated by some static store instructions are more (less) likely to be combined than those from others. Thus we have two types of predictors: a *MLI region* predictor and a PC-based *region payload* predictor. The former tracks the payload of recent MLIR messages and deactivates an MLI unit (*MLI_End*) when their payload is below a certain threshold. The latter tracks the payload of the MLIR messages when the corresponding message is brought in due to an IWDPR generated by a given store instruction, and determines whether a future invalidation request due to that store instruction should be an INV or IWDPR.

For the following set of results our default configuration is a homogeneous system with 8 cores, a MESI protocol, 1MB private caches, 4KB regions, and 32 buffers in each cache MLI unit.

Figure 6 presents the relative invalidation traffic with different predictors. Each benchmark has four bars: without a predictor (W/O), with an MLI region predictor (W Region), with a MLI payload predictor (W PC-Payload), and with both predictors. The percentage of the overhead (topmost portion of each bar) is annotated at the top of each bar. Notice that the overhead has been reduced significantly and that, with the combination of predictors, there is no degradation for any benchmark. Also notice that the number of times normal invalidation is used (the bottom portion of each bar) increases, especially in the cases where the overhead was high, because MLI was deactivated more often. However, as we shall see later, many of the invalidations are still conveyed by MLIR messages since each MLIR message conveys multiple invalidations.

MLI buffers in the cache MLI units for the 3 different cache sizes is 32, and the region size is 4Kbytes. Each bar in the figure has 4 components: the top is the delay permission overheads (MLIR messages with 0 payload). The next corresponds to MLI invalidation messages carrying out

Figures 7(a) and 7(b) shows the relative invalidation traffic for different numbers of buffers[1] (4, 16, and 32) in a cache MLI unit and for three different region sizes (1KB, 2KB, 4KB), respectively. The data suggests that an MLI unit does not need many buffers (16 or 32 seem adequate) and that 4KB (64 cache lines) regions are preferred for the benchmarks.

Figure 8 shows the relative invalidation traffic for multi banking directories (1, 2, and 4 banks) using *line-level* interleaving. The traffic reduction continues to be effective even with multiple banks. An important reason for MLIR effectiveness is combining invalidations from multiple consecutive regions into a single MLIR message (though this is not broken out in the figure).

We now consider the effectiveness of MLI messages in our example heterogeneous system. Figure 9 presents the relative invalidation traffic for a heterogeneous system that has 2 cores, 4 shader cores with 1MB private caches, 128B cache line, 8KB regions, and 64 buffers per MLI unit. Note the significant reduction (about 5X or greater) in invalidation traffic in 4 of the 5 benchmarks, more so than in the case of the benchmarks in the homogeneous system. This is because programming styles used for applications for heterogeneous systems result in similar operations being carried out on different data in large regions of code, characteristics that are well suited for MLI messages.

4.3 Characteristics of MLI Messages

We now consider characteristics of MLI messages in MLI regions, in our default homogeneous system. Figure 10 presents the average payload in an MLIR message (AVG) and two CDFs per benchmark, one (in dark gray) for the payload of an MLI message, and the other (in light gray) for the total number of invalidations carried out with MLI messages of a certain payload. The payload is the number of lines to be invalidated in an MLIR message. For the fluidanimate and canneal cases, showing low temporal and spatial locality of invalidations (data not shown in paper), we observe that the two CDFs are close, suggesting that most invalidations are being carried out small payload messages. Not surprisingly, there isn't a reduction in invalidation traffic. At the other extreme is barneshut, where 90% of the invalidations are carried out with MLIR messages with a payload of greater than 10 (60% with payload > 50), which is why there is a significant reduction in invalidation traffic.

In addition to the CDFs, each figure has two more numbers highlighted. The numbers concern the MLIR messages that reach a remote cache, whereas the CDFs are for MLI messages sent from a cache to the directory (not all MLIR

[1] The size of an MLI buffer is about 20 bytes for a system with 64B cache line, 4KB regions, and 42 bits address space.

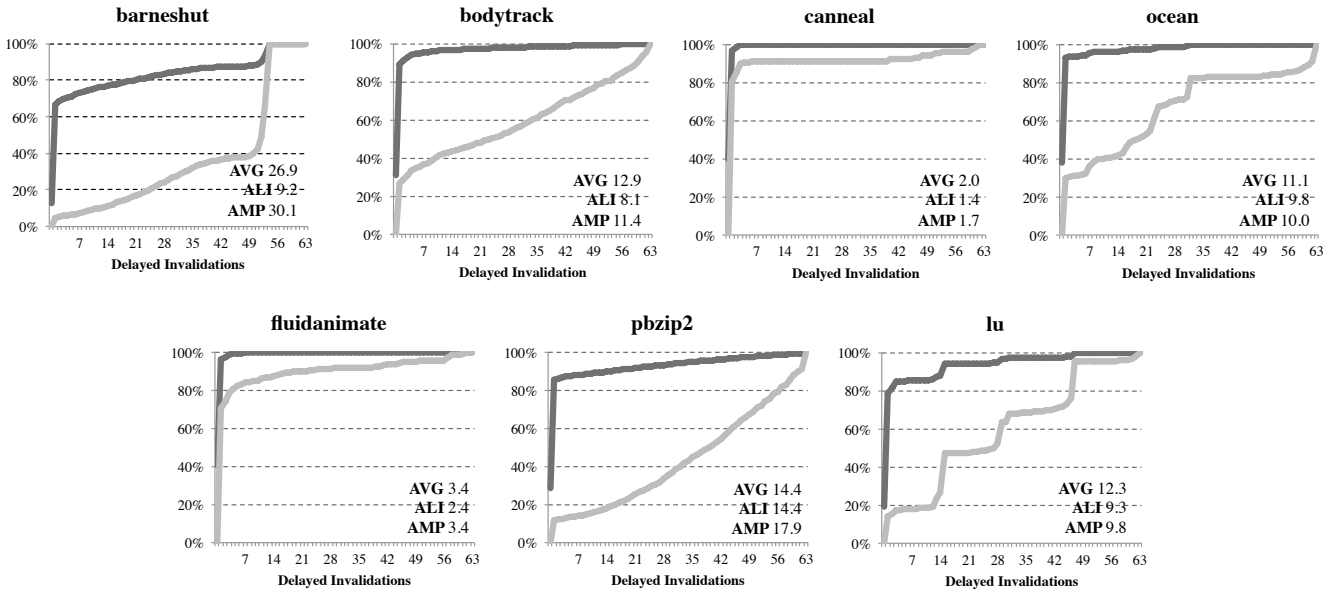


Figure 10. Characteristics of MLI Messages

messages to a directory result in MLIR messages to remote cache(s)). The first number is the *Average Message Payload* (AMP), which is the number of lines to be invalidated in an MLIR message that reaches a remote cache. The second number is the *Average Lines Invalidated* (ALI), which is the average number of lines in the remote cache that are actually invalidated in response to an MLIR message. The two numbers are very close in all cases, suggesting that most of the invalidations contained in an MLIR message perform an actual invalidation.

4.4 Reduction in Energy Consumption

The invalidation traffic reduction with MLI messages represents a decrease in an activity factor of the address network: fewer flits flow through the links, which also contributes to the reduced dynamic activity in the network nodes, with the commensurate reduction in dynamic energy. Invalidation traffic, however, may only be a small component of the overall traffic on the address network, which also includes read misses and write misses, as write-back activity. Hence, we discuss, in Table 4, the energy benefit from MLI messages by taking all the sources of traffic on the address network into consideration.

The first column shows the percentage of the address network traffic due to invalidations in the MLI regions without MLI messages; this is the amount of traffic that we can optimize. The next two columns present the number of invalidation messages per 100K micro-ops with and without MLI messages. As Martin, *et al.* [26], have observed, the invalidation traffic—the overhead of maintaining coherent caches—is not significant in some cases. But we are able to reduce this overhead even further. Importantly, in the cases where this overhead is significant (e.g., barneshut, ocean, bodytrack, streamcluster, and backprop), we are able to achieve significant reductions. There is little benefit for fluidaminate, canneal, kmeans, and lavaMD, but here the overhead is already low (few invalidations per 100K micro-ops).

	Benchmarks	% INV traffic in MLI regions	INV/100K micro-ops		Energy Reduction
			W/O MLI	With MLI	
Homogeneous Multiprocessor	ocean	20%	5.8	4.2	10.7%
	lu	4%	1.3	0.5	2.9%
	barneshut	32%	9.2	1.2	27.0%
	bodytrack	11%	3.3	2.3	7.1%
	pbzip2	6%	1.3	1.0	3.0%
	fluidanimate	2%	1.6	1.6	0.0%
	canneal	1%	2.7	2.7	0.0%
Heterogeneous Processor	streamcluster	18%	36.2	2.6	16.4%
	backprop	24%	27.9	5.6	18.6%
	bfs	8%	2.8	1.9	2.7%
	kmeans	1%	0.3	0.0	1.0%
	lavaMD	7%	0.2	0.0	6.5%

Table 4. Dynamic Energy Benefit with MLI Messages for overall Address Network

This traffic reduction translates into the decrease in the dynamic energy on the address network. The last column presents the energy reduction with MLI messages[2]. For example, for barneshut, the total address network energy with MLI messages is about 73% of the total energy of the baseline without MLI messages. The reduction is entirely due to reducing the invalidation traffic in an MLI region because other components of the traffic do not change (modulo variability in simulation). Workloads with a larger fraction of baseline invalidation traffic show more benefits.

[2] The energy consumption is derived from the product of execution cycles and dynamic power based on Orion power models [37].

4.5 Timing Results

We now consider the timing impact of MLI messages. They can have a positive impact on timing by reducing traffic (and contention) in the network as well as by saving the network latency for transferring invalidations. They can also have a negative impact since processing an MLI message at a directory/cache can take longer (possibly creating contention at these points) than a single invalidation message.

Table 5 presents the relative execution time of a system with MLI messages over the baseline for a homogeneous system[3]. MLI messages are charged an additional time, proportional to the payload, to process at a directory/cache. The lower contention column is our baseline configuration of earlier experiments. There is a slight performance benefit in cases where there is significant invalidation traffic (and thus MLI messages are effective). The higher contention column is for a scenario where the latencies in the network are increased, and thus there is more contention. Here a reduction in network traffic (and thus contention) permits a larger reduction in execution time. Importantly, the additional cycles at a remote cache/directory for a heavy-payload MLI message do not negatively impact performance.

The lack of more significant timing improvements with MLI messages should not be viewed as a negative. As with any microarchitectural technique, multiple factors are involved in determining the goodness of a technique. Many cache optimization techniques do not result in performance improvements, but are still useful because they improve other important parameters. For several cases where invalidation traffic is noticeable, MLI messages have a significant reduction in invalidation traffic (e.g., 5X), and a tangible improvement in the energy expended on the address network, in addition to the improvements in performance.

5. Other Related Work

In addition to the related work that we discussed in Section 2, there is a plethora of work in cache coherence and optimizations that exploit related phenomenon to the ones that we exploit in this paper. None of it appears to have exploited the combination of observations to optimize invalidation traffic as we have in this paper.

The early work on coarse-grain coherence techniques has been followed by additional work exploiting similar phenomenon. Zebchuk, *et al.*, propose a comprehensive framework (RegionTracker) for coarse-grain optimizations [40]. Zebchuk, *et al.*, also propose a tag-less coherence directory to optimize the storage requirement of directory-based coherence [41]. Enright-Jerger, *et al.,* use such behavior as the motivation for optimizing multicast operations [20]. Recent work by Cuesta, *et al.* [11], proposes to optimize coherence directories by deactivating coherence for private blocks, analogous to what coarse-grain schemes achieve in a broadcast-invalidate system.

[3] The timing results for a heterogeneous system are not presented since we haven't, as yet, integrated the Garnet network simulator with gem5-gpu.

Bench-marks	Lower Contention	Higher Contention
ocean	0.99	0.99
lu	1.00	1.00
barneshut	0.98	0.96
bodytrack	0.99	0.98
pbzip2	1.00	0.99
fluidanimate	1.00	1.00
canneal	1.00	1.00

Table 5. Relative Execution Time

Self-invalidation approaches have a different take on the timing of invalidation operations: rather than place the burden of invalidating copies in remote caches on the writer, it is the (prior) readers' job to self-invalidate their cached copies. In essence, the invalidation operations from the writer never need to happen (i.e., they are delayed for ever) since it is known that they will never find a matching block in a remote cache. Software cache coherence is the canonical example of self-invalidation where software takes responsibility for evicting (invalidating) lines from a private cache when it is likely that other processors might be writing to them [3][9][12][22][31]. Lebeck and Wood presented an early proposal for dynamic self-invalidation where hardware dynamically identifies blocks for self-invalidation [25].

While a lot of prior work assumed that programs were written in an arbitrary fashion, recent work exploits the observation that future parallel programs are likely to follow a more disciplined style of programming [6][10]. Choi, *et al.*, mention an important attribute of such a programming style: freedom from data races, and perhaps even guaranteed deterministic execution. Freedom from data races permits significant flexibility in the timing of invalidation operations: when code running on a processor is modifying a data structure, it is guaranteed that no other processor can access the associated cache lines. Thus, if remote copies of those cache lines exist, they need not be invalidated until a (much) later time when it is possible that code running on the remote processor(s) might access that data structure. Kaxiras, *et al.,* exploit data-race-free properties to propose an enhanced dynamic self-invalidation protocol—SARC coherence—where traditional coherence is carried out only amongst writers and transparent reads (whose corresponding line will be self-invalidated) do not downgrade the writer [21]. Choi, *et al*., exploit these properties to propose a coherence protocol for DeNovo, an architecture for a disciplined programming environment, which is simpler and generates a smaller amount of network traffic than a traditional MESI protocol [10]. The goal of these proposals is to develop new, possibly optimized, coherence protocols, enabled by significant demands placed on software, whereas our objective is to use similar phenomenon to optimize the messaging overheads of widely-deployed coherence protocols while not placing any additional demands on software. That is, we exploit the expected (but not guaranteed) case.

The property that programs access larger data structures, and thus read multiple contiguous cache lines, has been exploited to propose novel data prefetching and memory streaming techniques [35][38]. Other work has also proposed the use of bulk, coherent data transfers [18]. The phenomenon that data does not reside in multiple caches has also been exploited by techniques for predicting in which cache data might reside, i.e., destination set prediction [29]. The work on software distributed shared memory (of which the work cited in section 2 is an example) also exploits the ability to delay multiple messages to the same page, albeit using an entirely different approach in a different context.

6. Concluding Remarks

We proposed the concept of multi-line invalidation (MLI) messages to reduce the overheads of maintaining coherent caches in a multiprocessor system and demonstrated their effectiveness in reducing the number of invalidation messages, and in the energy usage on the address network, for a directory-based coherence protocol, for example homogeneous and heterogeneous multiprocessors. MLI messages can be successfully deployed along a normal line-based coherence protocol without changes to the coherence protocol or memory consistency model.

Future parallel programs are likely to employ a disciplined style where a significant portion of program's execution footprint is likely to be free of data races. Thus considerable opportunities for delaying invalidations will be prevalent. Combined with fact that greater parallelism will employ increasing degree of data decomposition—where different elements of larger data structures (in different, spatially-proximate cache lines) will see similar computation and memory access (and coherence) operations—the opportunities for deploying MLI messages to reduce coherence overheads are likely to be even more than for the existing workloads that we considered in the paper.

7. References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer,* 1996.

[2] N. Agarwal, *et al.*, GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. *In proc of* ISPASS 2009.

[3] J. Bennett, *et al.*, Adaptive software cache management for distributed shared memory architectures. In *Proc. of 17th ISCA*, 1990.

[4] Christian Bienia, Benchmarking Modern Multiprocessors. Princeton University, 2011.

[5] N. Binkert, *et al*., The gem5 simulator. *2011 SIGARCH Comput. Archit. News*.

[6] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proc. of 1st USENIX conference on HotPar,* 2009.

[7] J. F. Cantin, J. E. Smith, M. H. Lipasti, A. Moshovos, and B. Falsafi. Coarse-grain coherence tracking: Regionscout and region coherence arrays. *IEEE Micro,* 2006.

[8] S. Che, *et al.*, A Benchmark Suite for Heterogeneous Computing. In *Proc of IISWC*, 2009.

[9] H. Cheong and A. Veidenbaum. Compiler-directed cache management in multiprocessors. *Computer,* 1990.

[10] B. Choi, *et al*., Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proc. of 20th PACT,* 2011.

[11] B. Cuesta, A. Ros, M. E. Go´mez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proc. of 38th ISCA, 2011*.

[12] F. Dahlgren, *et al*., Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors. *JPDC*, 1995.

[13] E. Darnell, J. M. Mellor-Crummey, and K. Kennedy. Automatic software cache coherence through vectorization. In *Proc. of 6th ICS,* 1992.

[14] M. Dubois, *et al*., Delayed consistency and its effects on the miss rate of parallel programs. In *Proc. of 1991 Supercomputing.*

[15] M. Ekman, P. Stenstrom, and F. Dahlgren. TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *Proc. of 2002 ISLPED*.

[16] gem5-gpu.cs.wisc.edu

[17] J.Huang, What's Next in GPU Technology?. 2013.

[18] J. Heinlein, *et al*., Coherent Block Data Transfer in the FLASH Multiprocessor. In *Proc. of 11th IPPS,* 1997.

[19] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer Manuals, 2012.

[20] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *Proc. of 35th ISCA,* 2008.

[21] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling directory cache coherence in performance and power. *IEEE Micro,* 2010.

[22] L. Kontothanassis and M. Scott. Software cache coherence for large scale multiprocessors. In *Proc. of 1st HPCA,* 1995.

[23] Murali Kadiyala and Laxmi N. Bhuyan. A dynamic cache sub-block design to reduce false sharing. In *Proc. of 1995 ICCD*.

[24] P. J. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of 19th ISCA,* 1992

[25] A. Lebeck and D. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *Proc. of 22nd ISCA,* 1995.

[26] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM* 55, 7 (July 2012).

[27] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *Proc. of 32nd ISCA,* 2005.

[28] A. Moshovos, G. Memik, B. Falsafi, and A. N. Choudhary. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *Proc. of 7th HPCA,* 2001.

[29] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proc. Of 25th ISCA*, 1998.

[30] Phil Rogers. The programmer's guide to the apu galaxy. 2011.

[31] H. S. Sandhu, B. Gamsa, and S. Zhou. The shared regions approach to software cache coherence on multiprocessors. In *Proc. of 4th PPOPP,* 1993.

[32] C. E. Scheurich. Access ordering and coherence in shared memory multiprocessors. PhD thesis, University of Southern California Los Angeles, 1989.

[33] Inderpreet Singh, *et al.*, Cache Coherence for GPU Architectures, In *Proc of the 19th HPCA,* 2013.

[34] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence, Synthesis Lectures in Computer Architecture, *2011 Morgan & Claypool Publishers*.

[35] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proc. of 33rd ISCA,* 2006.

[36] D. Wallin, *et al.*, Bundling: reducing the overhead of multiprocessor prefetchers. In *Proc of 18th IPDPS*, 2004.

[37] H.-S. Wang, X.-P. Zhu, L.-S. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In *Proc. of 35th IEEE Micro*, 2002.

[38] T. F. Wenisch, *et al*., Temporal streaming of shared memory. In *Proc. of 32nd ISCA,* 2005.

[39] S. C. Woo, *et al*., The SPLASH-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit*, 1995.

[40] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *Proc. of 40th IEEE Micro,* 2007.

[41] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *Proc. of 42th IEEE Micro,* 2009.