An Evaluation of Snoop-Based Cache Coherence Protocols

Linda Bigelow Veynu Narasiman Aater Suleman

ECE Department
The University of Texas at Austin
{bigelow, narasima, suleman}@ece.utexas.edu

I. INTRODUCTION

A common design for multiprocessor systems is to have a small or moderate number of processors each with symmetric access to a global main memory. Such systems are known as Symmetric Multiprocessors, or SMPs. All of the processors are connected to each other as well as to main memory through the same interconnect, usually a shared bus.

In such a system, when a certain memory location is read, we expect that the value returned is the latest value written to that location. This property is definitely maintained in a uniprocessor system. However, in an SMP, where each processor has its own cache, special steps have to be taken to ensure that this is true. For example, consider the situation where two different processors, A and B, are reading from the same location in memory, and therefore, both processors have a copy of that data in their respective cache. As long as both processors are just reading the data, everything should be fine. However, if processor A wishes to write to that location, processor B needs to somehow be aware of that write; otherwise, the next time processor B reads from that location, it could read stale data. Therefore, in SMPs where each processor has its own cache, there is a need to keep the caches *coherent*. Mechanisms for keeping caches coherent in a multiprocessor system are known as *cache coherence protocols*.

One solution to the cache coherence problem is to allow processors to be able to observe, or *snoop*, the reads and writes of other processors when necessary. Solutions that employ such a mechanism are known as *snoop-based cache coherence protocols*. These protocols can only be used in systems where multiple processors are connected via a shared bus. Therefore, snoop-based cache coherence protocols are ideal candidates for solutions to the cache coherence problem in SMPs.

Despite solving the cache coherence problem, snoop-based cache coherence protocols can adversely affect performance in multiprocessor systems. For example, in uniprocessor systems, when a store is issued to a location that is present in the cache, in general, the write can proceed without any delays. However, in multiprocessor systems, even though the block may be present in the processor's cache, it may need to broadcast the write on the bus so that other processors can snoop the write and take appropriate action, which can significantly hurt performance. Therefore, the best snoop-based cache coherence protocol is one that not only maintains coherence, but does so with minimal performance degradation.

In the following sections, we will first describe some of the existing snoop-based cache coherence protocols, explain their deficiencies, and discuss solutions and optimizations that have been proposed to improve the performance of these protocols. Next, we will discuss the hardware implementation considerations associated with snoop-based cache coherence protocols. We will highlight the differences among implementations of the same coherence protocol, as well as differences required across different coherence protocols. Lastly, we will evaluate the performance of several different cache coherence protocols using real parallel applications run on a multiprocessor simulation model.

II. SURVEY OF EXISTING CACHE COHERENCE PROTOCOLS

Over the years, several cache coherence protocols have been proposed in the literature. In an early work by Archibald et al. in [1], six different coherence protocols were described, and their performance differences were compared using a multiprocessor simulation model. These six coherence protocols can be classified into two main categories: invalidation-based protocols, and update-based protocols. Invalidation-based protocols invalidate all other cached copies of the data on a write, whereas update-based protocols broadcast the write so that other cached copies can be updated. The four invalidation-based protocols studied in this work were Synapse, Goodman's Write-Once, Berkeley-Ownership, and Illinois. The two update-based protocols were Firefly and Dragon.

A. *Invalidation-Based Coherence Protocols*

Before going into the details of each of the invalidation-based protocols, a description of one of the simplest invalidation-based protocols, the MSI protocol, is necessary. The other protocols will then be discussed, and their differences with the MSI protocol will be highlighted.

Figure 1 on the next page graphically depicts the MSI protocol. There are three states in this protocol: M (Modified), S (Shared), and I (Invalid). A processor having a cache block in the Modified state implies that it is the only processor that has this block in its cache and that the data is not consistent with memory. The Shared state implies that the

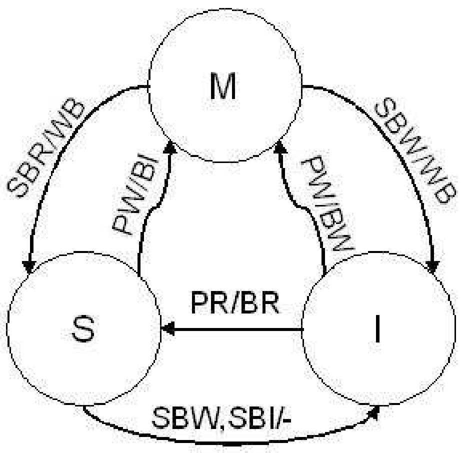

Fig. 1. MSI state diagram.

data is in the cache, but may also be present in the caches of other processors. The data is consistent with memory for the Shared state. The Invalid state implies that the block is not in the cache.

In this protocol, there are four possible bus transactions:

- Bus Read (BR): Occurs when a processor wishes to read from a cache block that is not in its cache. The block must be transferred into the cache from memory.
- Bus Write (BW): Occurs when a processor wishes to write to a cache block that is not in its cache. The block must be transferred into the cache from memory.
- Bus Invalidate (BI): Occurs when a processor wishes to write to a cache block that is in its cache, but that block may be present (shared) in the caches of other processors. The block need not be transferred from memory. This is also commonly called a Bus Upgrade.
- Write Back (WB): Occurs when a modified cache block needs to be written back to memory.

There are five possible actions that may cause a state transition:

- Processor Read (PR): A processor issues a load instruction.
- Processor Write (PW): A processor issues a store instruction.
- Snoop Bus Read (SBR): A processor observes a Bus Read operation.
- Snoop Bus Write (SBW): A processor observes a Bus Write operation.
- Snoop Bus Invalidate (SBI): A processor observes a Bus Invalidate operation.

In Figure 1, there is an arrow from the Invalid state to the Modified state labeled "PW/BW." This means that if a cache block is in the Invalid state and then the processor issues a store (PW) to that block, the Bus Write (BW) bus transaction will occur, and the local state of the cache block will be changed to Modified. Likewise, other processors will snoop the Bus Write bus transaction, and respond appropriately. If another processor has that block in the Shared state, it would simply change its state to Invalid. However, if a processor had the block in the Modified state, in addition to changing the state to Invalid, it would also have to write the block back to main memory. These two transitions are also depicted in the figure. Note that only events causing state transitions or bus operations are depicted in Figure 1. For example, if a cache block is in the Shared state and the processor snoops a Bus Read transaction or issues a load instruction, neither a bus transaction nor a state change is necessary. In order to simplify the figure, these events are not shown.

Notice that if a processor issues a store instruction to a cache block that is in the Shared state, a Bus Invalidate transaction occurs. Other processors that also had that block in the Shared state would snoop that Bus Invalidate, and change the state to Invalid. The writing processor would then change the state to Modified and perform the write. This behavior is at the soul of all invalidation-based protocols.

Lastly, when a block is chosen as a victim and needs to be evicted from the cache, a Write Back must be done only if the cache block was in the Modified state. In the following subsections, we give a description of the other invalidation-based protocols in relation to the MSI protocol.

1) Synapse: The Synapse protocol is actually a simplification of the MSI protocol described above. This protocol also has three states: Invalid, Valid (analogous to Shared in MSI), and Dirty (analogous to Modified in MSI). There are two main differences between Synapse and MSI. One is the transition that occurs when a Bus Read is snooped in the Modified state. Both protocols call for a Write Back, but in MSI, the state is changed to Shared, whereas in Synapse, the state is changed to Invalid. The other difference is that in Synapse, there is no Bus Invalidation transaction, there is only a Bus Write. Therefore when a block is in the Valid state and a store is issued to that block, a Bus Write, instead of a Bus Invalidate occurs, and a transfer from memory is initiated even though it is unnecessary. This difference will definitely hurt performance since the writing processor will have to wait for the entire cache block to be transferred before it can proceed. In addition, unnecessary bus traffic is created.

2) Goodman's Write-Once: Goodman's Write-Once protocol introduces a fourth state called the *Reserved* state. It also replaces the Bus Invalidate transaction of the MSI protocol with a Write Through transaction. In a Write Through transaction, the bus is acquired, and a word is written through to main memory. The only time a Write Through is initiated is when a cache block is in the Valid state and the processor issues a store to that block. The writing processor transitions to the Reserved State. All other processors that snoop the Write Through transit to the Invalid state, just as they did in MSI when they snooped a Bus Invalidate. Therefore, when a single processor has a block in the Reserved state, all other processors must have that block in the Invalid state, and the data is consistent with memory. A cache block in the Reserved state transitions to the Dirty state on a Processor Write without any bus transaction. In addition, a cache block in the Reserved state does not have to be written back to main memory on a replacement. The inclusion of the Reserved state and the Write Through bus transaction improves performance in the case where after reading from a cache block, a processor only writes to that block once. If this block needs to be replaced, a Write Back does not have to occur, whereas in the MSI protocol, a Write Back would be needed.

3) Berkeley-Ownership: This is also a four-state protocol, but the new state is a *Shared-Dirty (Owner)* state. In the MSI protocol, if a processor had a cache block in the Modified state and it then snooped a Bus Read (SBR) to that block, a Write Back would occur, and the state would change to Shared. However, in the Berkeley protocol, the processor supplies the data directly to the requesting processor and transitions into the Shared-Dirty state, thereby avoiding the costly write back to memory. This is the main advantage of the Berkeley protocol. Write Backs are only needed when a block in either the Dirty or Shared-Dirty state is evicted. Notice that this protocol allows for one processor to have a cache block in the Shared-Dirty state while other processors have the block in the Shared state. This implies that a block in the Shared state may have data that is not consistent with memory. This protocol is also known as the MOSI protocol, referring to the four possible states of a cache block: Modified (same as Dirty), Owner (same as Shared-Dirty), Shared (same as Valid), and Invalid.

4) Illinois: This is another four state protocol that behaves almost identically to the MSI protocol described earlier except for the inclusion of a new state, known as the *Exclusive* state. In the MSI protocol, if a read miss occurred, a Bus Read transaction would be issued, and the cache block of the requesting processor would always transition to the Shared state. However, in the Illinois protocol, the cache block of the requesting processor would transit to either the Shared state or the new Exclusive state, depending on whether or not the cache block is actually shared. If another cache had a copy of the data in its cache, then a transition to the Shared state would occur; otherwise, a transition to the Exclusive state would occur. Therefore, a processor that has a cache block in the Exclusive state implies that it is the only processor that has the data in its cache, and the data is consistent with memory (has not been modified). Once in the Exclusive state, if the processor wants to write, it can do so without a bus transaction; it just needs to transit to the Modified state. This saves unnecessary Bus Invalidate transactions that the MSI protocol suffers from. This protocol is also known as the MESI protocol, referring to the four possible states of a cache block: Modified (same as Dirty), Exclusive, Shared (same as Valid), and Invalid.

5) MOESI: Although not described by Archibald et al. in [1], the MOESI protocol is another common invalidation-based protocol. The MOESI protocol, as the name may suggest, is a five-state protocol that incorporates elements of both the MESI (Illinois) and MOSI (Berkeley-Ownership) protocols. It extends the basic MSI protocol with two new states: an Exclusive state, which is identical to the Exclusive state of the MESI protocol, and an Owner state, which is identical to the Owner state of the MOSI protocol. Including both an Exclusive and an Owner state enables the MOESI protocol to enjoy the benefits that both the MESI and MOSI protocols offer. However, this comes at the cost of increased complexity and additional storage bits to keep track of the state.

B. Update-Based Coherence Protocols

Update-based coherence protocols differ from invalidation-based protocols in the sense that Bus Invalidations are not needed when a processor wishes to write to a shared block. Rather, the processor broadcasts the write so other processors can pick up the new data and update their cache accordingly. The two most common update-based coherence protocols are Firefly, and Dragon.

1) Firefly: Firefly is an update protocol that uses Write Through (like the Goodman Write-Once scheme) instead of Bus Invalidates. It is a four-state protocol (Dirty, Exclusive, Shared, and Invalid); however, in most descriptions the Invalid state is left out since no bus transaction can ever cause another processor's cache block to be invalidated (this is possible in invalidation-based protocols). In the Firefly protocol, when a processor wishes to write to a block that is in the Shared state, a Write Through occurs, and memory is updated just as in the Goodman Write-Once scheme. However, in the Goodman scheme, all other processors that snooped the Write Through changed the state of their cache blocks to Invalid, whereas in Firefly, these processors would simply update their copy of the cache block and remain in the Shared state. This is what makes the Firefly protocol an update-based protocol and not an invalidation-based protocol.

2) Dragon: The Dragon protocol is similar to the Firefly protocol described above, except for the fact that it gets rid of the Write Through bus transaction and instead uses a Bus Update transaction. A Bus Update transaction broadcasts the write so that all other processors that have the data in their cache can update the cache block, but it does not go through to main memory as in the Firefly protocol. For this to work, the Dragon protocol has two shared states, *Shared-Clean*, and *Shared-Dirty*, in addition to the Dirty and Exclusive states. If there are multiple processors that are writing to the same block, the one that wrote to the block the latest would be in the Shared-Dirty state, and all others would be in the Shared-Clean state. However, every processor will have the most up to date version of the cache block since they snoop the Bus Updates of other processors and update their cache accordingly. A cache block needs to be written back to main memory on replacement only if it is in either the Dirty or Shared-Dirty state.

C. Performance Comparison of the Protocols

Archibald et al. in [1] compared six cache coherence protocols using a simulation model and found dramatic differences in their performance. One of the main reasons for the performance differences is the way in which the protocols handle reads and writes to private (not shared) blocks. Only three of the protocols used in this study (Illinois, Firefly, and

Dragon) have the ability to detect whether or not a cache block is truly shared when it is read from memory. In these protocols, if no other processor has the data in their cache, a transition to the Exclusive state is made, indicating that the processor has the only copy of the block and that the block has not been modified. Now, when the processor wishes to write to this block, it can do so without any bus transaction since it knows that no other processor has the data in its cache. The other three protocols do not have an Exclusive state and therefore would need to issue a bus transaction before proceeding with the write. Since a significant portion of all memory references are actually to private blocks, this difference can have a major impact on performance. For this reason, the Illinois, Firefly, and Dragon protocols usually performed the best in this study.

Another interesting result of this study showed that, in general, the update-based protocols (Firefly and Dragon) outperformed the invalidation-based protocols in terms of their handling of shared cache blocks. However, we feel that this claim is not well-grounded and is a result of an inaccurate simulation model. The model was probabilistic, and whenever a memory reference was due, it was randomly chosen to be either a reference to a shared block (with probability *shd*) or a private block (with probability 1-*shd*). The problem is that when using this model, it is very unlikely for one processor to read and write to a shared block for some time, then have another processor take over and exclusively read and write to that block, etc. This is known as *sequential sharing* and is quite common in many parallel applications. Invalidation-based protocols are known to perform better in these types of situations. On the other hand, when reads and writes to a shared block are interleaved among many processors, update-based protocols are known to perform better. The model used in this study is biased towards the interleaved scenario rather than the sequential sharing scenario, and therefore, the update protocols appear to perform better. Using real multiprocessor applications would lead to different results. Despite this drawback, this study still provides several good insights concerning the different cache coherence protocols that exist and the potential impact on performance that they may have.

D. Improvements to Invalidation-Based Coherence Protocols

Eggers et al. in [5] described the major limitations and problems that invalidation-based protocols suffer from. Improvements to the coherence protocols are suggested and their impact on performance is presented.

In invalidation-based protocols, consider the case where a cache block is shared among many processors. Reads to this cache block can occur locally without any bus transactions. However, when one of the processors wishes to write to that block, a Bus Invalidation signal is sent out, and all other processors change the state of the cache block to Invalid. If any one of these processors wishes to read a value from this block in the future, it will incur a read miss and have to request the data from memory. These types of misses are called *invalidation misses* and considerably hurt the performance of all invalidation-based protocols. Eggers et al. in [5] also pointed out that this problem gets worse as the cache block size increases. With a larger cache block size, the probability that one processor writes to a part of a block that is shared by many other processors (these processors may be reading from a different part of the block) is increased.

A possible solution to this problem was proposed and is known as *Read-Broadcasting*. Consider the same situation described earlier where many processors are sharing a cache block and then one of them wishes to write to the block. Just as before, all other cache blocks will be invalidated. However, when one of these recently invalidated processors wishes to read a value from this cache block, all of the other processors will snoop that read, grab the data from the bus, and transit into the Shared state. This will only take place if the block that was invalidated is still invalid and has not been replaced by another block. This implies a new transition from the Invalid state to the Shared state when the processor snoops a Bus Read. Now, when the other processors also read a value from this cache block they will get a cache hit instead of a miss, thereby reducing the number of invalidation misses.

The results showed that adding the Read-Broadcast extension to an invalidation-based protocol (they used the previously described Berkeley Ownership protocol for this study) did indeed reduce the number of invalidation misses, but did not improve overall performance. The reason for this is that allowing snoop matches in the Invalid state can lead to increased *processor lockout* (this will be explained in more detail later) from the cache, thereby hurting performance. If this offsets the gain from reducing the invalidation misses, net performance remains the same. In fact, in many of the applications used, a slight decrease in overall performance was obtained, indicating that the disadvantage of increased processor lockout more than offset the improvement obtained from reducing the number of invalidation misses. This illustrates an important point when analyzing the performance of coherence protocols: it is not enough to simply track bus traffic or the number of cache misses since other factors, such as processor lockout from the cache, can have an offsetting impact on overall performance.

E. Improvements to Update-Based Protocols - Hybrid Cache Coherence Protocols

We have noted that the performance of invalidation-based and update-based protocols depends highly on the application. As mentioned earlier, for applications where data is sequentially shared, invalidation-based protocols outperform update-based protocols. However, for applications where reads and writes from different processors are interleaved, the update protocols outperform the invalidation based protocols. Given this, the next logical step is to come up with a hybrid protocol. Hybrid protocols use an invalidation-based protocol when it is best to do so, but can also use an update based protocol when it is best to do so. Veenstra et al. in [8] performed a study of the potential performance improvement that could be obtained from using a hybrid protocol. Specifically, an application was run using only an invalidation-based protocol and then also run using only an update based protocol. During the runs, statistics for each cache block were recorded, and a cost metric for each cache block was computed. The cost metric took into consideration the number of cache hits, invalidations or updates, and the number of times the block had to be loaded into the cache. Each of these factors was weighted corresponding to the number of cycles they would take (for example loading a block is much more costly than a cache hit). The cost for a block when using an invalidation based protocol was compared to the cost for the same block using an update-based protocol. The protocol with the lower cost was determined to be the better one for that cache block. The application was run again, but this time the coherence protocol for each block was whichever one was determined to be better for that block. The results showed that hybrid protocols can significantly improve performance over pure invalidation-based or update-based protocols. Clearly a one time static decision for the coherence protocol of a cache block is not a realistic option since different applications will have completely different results. Nevertheless, this study effectively illustrated the potential improvements that can be achieved using a hybrid protocol.

Two actual hybrid protocols are discussed by Dahlgren in [4]. *Competitive Snooping* is an update-based protocol that can dynamically switch to an invalidation-based protocol. When a cache block is loaded into the cache for the first time, an update counter associated with the block is initialized to a certain threshold value. Whenever the processor snoops an update to the block, the counter is decremented. However, whenever the processor actually references the block (reads from or writes to it), the counter is set back to the initial threshold value. Upon snooping an update, if the counter value is zero, the cache block is invalidated. A counter value of zero implies that this processor was not actively using the block and, therefore, will likely not reference the block again. Invalidating such a block could improve performance since the processor actually writing to the block could detect when all other processors that are not a[illegible]ly using the block have been invalidated. At that time, the writing processor could transit to the Modified state, and all future writes to that block would be local, thereby avoiding several Bus Update transactions. This protocol involves considerably more hardware support, and also the use of a *shared* bus control line so that a processor can detect when no other processor has a cache block in its cache.

The *Archibald Protocol* improves the Competitive Snooping technique described above by making additional use of the shared control line. Consider the case where multiple processors all have a particular cache block in their cache, but only one of these processors is actively referencing the block. All other processors will snoop the updates from the writing processor and decrement their counters. In Competitive Snooping, as soon as the counter for a cache block becomes zero, the block is invalidated. The Archibald Protocol observed that it is not useful to immediately invalidate the block. A block will be invalidated only when all of the inactive blocks of other processors are also ready to be invalidated (their counters have also reached zero). To implement this, when a processor snoops a Bus Update, it decrements the counter and asserts the shared control line only if the counter value is non-zero. With this policy, after a Bus Update that decrements the last inactive block to zero, no processor will assert the shared control line (since all of their counters are zero). All of the inactive blocks will know that they should now be invalidated, and the writing processor will transition to the Modified state so that future writes can be local.

Hybrid protocols have shown improvement over pure invalidation-based or update-based protocols in terms of the number of bus transactions, the bus traffic, and the number of coherence related misses. However, to our knowledge, they have not been widely adopted in real multiprocessor systems. Perhaps the reason for this is that although they do reduce bus traffic and the number of coherence-caused misses, they can also lead to increased processor lockout, thereby mitigating the overall performance improvement. Also, the inclusion of counters in the tag store leads to added complexity, larger area, and more power consumption. The mitigated performance improvements obtained may not justify this increase in area and power.

F. Energy Efficiency of Cache Coherence Protocols

Power has become an increasingly important metric in today's processor design. Loghi et al. in [7] analyzed the differences in power consumption of various cache coherence protocols. However, the results of this paper are rather disappointing. The major conclusion drawn from this study was that the cache write policy (write through or write back) was more important than the actual cache coherence protocol in terms of affecting total energy consumption. The study showed that coherence policies that used a write-through policy consumed significantly more energy than those that used a write-back policy. This result is not surprising at all and is an obvious result of the fact that a write-back policy leads to fewer accesses to main memory. In addition, only two coherence protocols that use the write-back policy were analyzed, the Berkeley-Ownership protocol and the MESI protocol. The results showed that the energy numbers produced by these two protocols were virtually identical for all of the applications. It would have been instructive to see some comparison between the energy consumption of invalidation-based protocols, update-based protocols, and hybrid protocols. However, no such comparison was made. With the increasing importance of low-power design, future studies comparing the energy and power consumption of different cache coherence protocols would be very helpful.

III. IMPLEMENTATION CONSIDERATIONS

To better understand the relative efficiency and benefit of each of the cache coherence protocols, it is important to consider the underlying hardware that would implement each of the protocols. Specifically, the design of the Bus Interface

Unit (BIU) will have a significant impact on the performance of the cache coherence protocol. In the following sections we give a general overview of the BIU and then discuss both coherence protocol independent and dependent design points.

A. Bus Interface Unit Overview

The BIU is responsible for providing communication between a processor and the external world (i.e. memory and other processors) via the system bus [6]. Although the design of the BIU is dependent on the bus protocol and cache protocol that are used, there are some general functions that any BIU should perform. First, the BIU is responsible for arbitrating for the bus when its associated processor needs to use the bus. If the system uses separate address and data buses, the BIU may need to arbitrate for just the address bus (in the case of a read) or both the address and data buses (in the case of a write). After the BIU requests the bus, it must wait for the bus grant signal and then drive the address and control lines for the particular request [3].

The BIU is also responsible for controlling the flow of transactions and information to and from the bus. To handle requests for bus transactions from the processor's cache, the BIU maintains a (usually) small request buffer, which may issue requests onto the bus in a first-in, first-out (FIFO) order or according to some predetermined priority. Once the system has responded to the request, the BIU must communicate the response information back to the cache. In the case of a read request, the incoming data would be taken off the bus and placed in a (usually) small response buffer. As part of maintaining the buffers, the BIU is also responsible for taking any necessary actions to handle a request or response that is received when the associated buffer is full.

Finally, the BIU is responsible for snooping the bus transactions initiated by other processors. When the BIU snoops a read or write to a particular address, it must first determine if it is an address that is also present in its own processor's cache by looking up the tag in the tag store. Then, depending on the type of transaction (read, write, etc.), whether or not there was a tag match, and the cache coherence protocol in use, the BIU must respond appropriately. This response may be asserting a shared signal (in the case of snooping a read request to a shared cache block for a MESI protocol), updating a cache block (in the case of snooping an update operation to a shared cache block in the Dragon protocol), writing back a dirty cache block (in the case of snooping a Bus Read operation to a modified cache block in the MESI protocol), etc.

B. Coherence Protocol Independent Design Considerations

There are several design issues for the BIU that affect overall system performance and efficiency regardless of the particular cache coherence protocol that is used. Some of these issues include access to the tag store, the presence of a write-back buffer, and cache-to-cache transfers.

1) Tag Store Access: Since the BIU is responsible for snooping bus operations and comparing the address on the bus to addresses in its own cache, the BIU must have access to the tag store. If the tag store is single-ported, then the BIU will be competing with the processor for access to the tag store [3] [1]. Whenever the BIU is accessing the tag store, the processor will be locked out, and whenever the processor is accessing the tag store, the BIU will be locked out. If there is a lot of contention for access to the tag store, then both the processor occupancy and the response latency of the BIU will be negatively impacted. Although it is possible to always give the processor priority over the BIU, doing so may decrease the effective bus bandwidth due to increased latency. Conversely, always giving the BIU priority may decrease processor performance due to increased processor lockout. Since there is no obvious performance win, determining which device should be given priority must be decided within the context of the overall system design goals.

To reduce the contention for the tag store, the tag store can either be dual-ported or duplicated. If the tag store is duplicated, one copy will be used by the BIU for looking up snooped addresses, and the other copy is used by the processor. Since the BIU only needs information about a cache block's tag and its state in the cache coherence protocol, it is not necessary to duplicate all of the information in the tag store for the BIU's copy; providing only the tag and state should be sufficient. Duplicating the tag store allows the BIU and the processor to read and lookup tags simultaneously. However, when the tag or state for a block needs to be updated (due to a bus invalidation, a cache eviction, etc.), both copies of the tag store must be modified. This means that the BIU or the processor may still need to be locked out temporarily. Duplicating the tag store can also be quite costly in terms of the area required to hold the duplicate copy.

Providing a dual-ported tag store would require less area than duplicating the tag store and would still allow the processor and BIU to lookup tags simultaneously. Additionally, maintaining a consistent view of the tag store between the processor and the BIU would not be a problem as it was in duplicating the tag store. However, even with a dual-ported tag store, the processor or the BIU will still need to be locked out when the tag store must be updated. Adding another port may also not be desirable from a hardware design complexity or power consumption standpoint.

Another consideration in the decision to duplicate (or dual-port) the tag store is the design of the cache hierarchy. If the system supports a multilevel cache hierarchy, as most do, at which point in the hierarchy should the tags be duplicated? As noted in [3], dual tags are actually less critical in a multilevel cache hierarchy. Consider a two-level cache hierarchy in which the L2 maintains inclusion of the L1. It is then possible for the BIU to lookup tags in the L2 tag store in response to bus snoops, whereas the processor should be doing most of its lookups in the L1 tag store. Care must be taken to ensure that any necessary updates due to bus snoops are passed from the L2 up to the L1 and that any modifications to data made by the processor in the L1 are passed down to the L2.

There are some cases where the snooped bus operations may apply to data that is in the L2 but not in the L1; however, if the data is also in the L1, then the action in response to the snoop (eg. an invalidation signal or a flush) needs to be propagated up to the L1. One way to do this is to pass every snooped transaction up to the L1 and let the L1 decide, based on the address, if it needs to do anything about it. The downside to this approach is that there may be a lot of unnecessary traffic in the L1, which can hurt performance. Another approach is to keep a single bit associated with every block in the L2 that indicates if this block is also in the L1. If the block is in the L1, then the L2 must pass up the bus transaction; however, if the block is not in the L1, then there is no need to bother the L1. In this sense, the L2 acts as a filter for the L1 for bus transactions, which leaves the L1 tags available for the processor to use, thereby reducing processor lockout.

The only time that the L1 must pass information down to the L2 is when the processor issues a write to a cache block that hits in the L1. In this case, the L1 must send the data to the L2 so that the L2 has the most recent copy of the data for any bus transactions associated with that cache block. One way to do this is to make the L1 write-through to the L2. Another option is to make the L1 a write-back cache, but have it communicate a bit of information to the L2 indicating that a block was just written to in the L1, and therefore, should be marked *modified-but-stale* in the L2 [3]. Then, if there are any bus transactions that involve this data, the L2 will see that it has a stale copy of the data and forward the request to the L1. Ideally, the L1 will be able to satisfy most of the processor's requests and leave the L2 for the BIU to use, thereby reducing BIU lockout.

2) Write-Back Buffer: Another important consideration for the efficiency of the BIU and system performance is how write-backs are handled. Since write-backs due to snooping bus writes result in both the dirty data being written to memory and the data being supplied to the requestor, two bus transactions may be required. Suppose, for example, that processor A has a cache block in the Modifed state, and processor B has the same cache block in the Invalid state. When processor B issues a write to that cache block, it will miss in processor B's cache and generate a request for the block. In the meantime, processor A will snoop the write to the cache block. Processor A should then issue a write-back to memory, and memory should supply the cache block to processor B. Ideally, we would like processor B to be able to continue with its work as soon as possible. This means that we would like to service the write miss first and delay the write-back [3]. To do this, processor A's BIU needs to maintain a write-back buffer to hold the delayed data value. After the write miss is satisfied (processor A supplies the data to processor B), then processor A's BIU may arbitrate for the bus and send the value in the write-back buffer to memory.

When incorporating a write-back buffer, it is important to consider what should happen if a bus transaction occurs for an address that has not yet been written back to memory from the buffer. Now, in addition to looking up the address in the tag store, the BIU must also check the write-back buffer for an address match [3][6]. If the address in question is found in the write-back buffer, then the BIU should supply the value from the write-back buffer in response to the bus transaction and cancel the pending write-back (i.e. invalidate the entry in the write-back buffer). This type of implementation assumes that whoever receives the modified data from the write-back buffer will take the responsibility of eventually writing back the value.

3) Cache-to-Cache Transfers: To improve the timeliness of requests for data that is cached somewhere, it may be useful to allow another cache to supply the data rather than always going to memory. To support cache-to-cache transfers, each BIU must have a way of indicating whether or not its cache has the data and is able to supply it. This can be accomplished by means of a single wired-OR bus control line, which is asserted by each BIU if its cache is able to supply the data. The memory controller will also know that it should not supply the data if this control line is asserted. If several caches are able to supply a copy of the data, there must either be a way of selecting one of those caches (by using a priority or ownership scheme, for example) or it must be guaranteed that all caches will put the same value on the bus. In [1], it is noted that this additional complexity and the potential for slowing down the processors that have to provide the data from their cache has resulted in a limited use of cache-to-cache transfers.

Another issue with cache-to-cache transfers is whether or not one of the caches has the cache block in question in a Modified state, which can be indicated by another wired-OR bus control signal asserted by the BIU. If one of the caches has a modified copy of the block, it should not only be the cache that is chosen to supply the data, but it should also inhibit memory from supplying the data since memory has a stale copy [1][3]. As the cache is supplying the modified data to the requestor on the system bus, it is possible to write the modified value back to memory at the same time (as opposed to writing it back in a separate bus transaction). While this saves a bus transfer and improves bus performance, it has the additional complexity of having three cooperating members on the bus [1].

C. Coherence Protocol Dependent Design Considerations

Depending on which cache coherence protocol is used, there are additional modifications to the basic BIU design that may be necessary. In the case of the MESI protocol (or any other protocol that has an exclusive state), when there is a read miss in the cache, the coherence protocol may transition to one of two states: shared or exclusive. Since this transition is based on whether or not another cache has a copy of the same block, there must be a mechanism for indicating that a block is shared. Typically this is done by providing a *shared* bus control line that is a wired-OR of the shared signals coming from each of the BIUs. In protocols that include an owner state, such as MOSI and MOESI, the BIU must be able to support the cache-to-cache transfers mentioned in the previous section and inhibit memory from responding to read requests when appropriate. Finally, there is a difference in the BIUs behavior for invalidation-based versus update-based protocols. In invalidation-based protocols, the BIU only needs to grab data off of the data bus when its processor has a read miss; however, in update-based protocols, the BIU also needs to be aware of updates that are broadcast on the bus and grab the data off the bus if its cache has a copy of that particular cache block.

IV. EXPERIMENTAL METHODOLOGY

We used the SPLASH-2 benchmark suite [9] to evaluate the performance of four cache coherence protocols by simulating their execution on the M5 simulator [2].

A. The M5 simulator

The M5 simulator can be operated in full system mode or in system emulation mode. In system emulation mode, system calls are not actually simulated, but instead are emulated on the host machine. We are currently operating the simulator in system emulation mode to reduce the simulation time. However, if time allows, we intend to run some of the experiments in full system mode. The M5 simulator uses an interface that heavily relies on the python programming language. Most of the parameters are specified in python configuration files, which are used to trigger the simulations. We modifed the sample configuration files that were included in M5 to customize the simulator to our needs. In system emulation mode, M5 can be configured to use cycle accurate and detailed processors or simple functional models. These processor models are called Detailed and Simple, respectively. We experimented with both the Detailed Model and the Simple Model. The Detailed model provides more data regarding the simulation and also allows us to specify more parameters in the system. The downside of using the Detailed model is that it is close to 20 times slower than using the Simple model. To maintain accuracy and at the same time save simulation time, we chose some benchmarks at random and compared the results of both the Detailed and Simple models. We noticed that the statistics concerning the coherence protocol (such as number of Bus Invalidate Signals, etc.) collected using the Simple model were identical to the statistics collected using the Detailed model. Therefore, we decided to use the Simple model to collect the data related to the coherence protocols by running the benchmarks to completion. However, we could not extract certain important metrics from the Simple model, such as the IPC. Therefore, we also used the Detailed model to get an estimate of IPC numbers for each benchmark by simulating only the first 10 million instructions of each benchmark. We have not yet simulated all the benchmarks to completion using the Detailed model, but we plan to do so in the near future.

B. Coherence Protocols

We have simulated four cache coherence protocols, namely the MSI, MESI, MOSI, and MOESI protocols. M5 gives us the option to treat Bus Invalidation transcations (also known as Bus Upgrade, or Get Permission) as Bus Write transactions. Unlike Bus Invalidations, Bus Writes involve the transfer of an entire cache block from memory. In many cases, all that is needed is an Invalidation transaction, so replacing them with Bus Writes, although functionally correct, hurts performance. Even though some implementations (such as in the Synapse protocol) do not distinguish between Bus Invalidates and Bus Writes, we chose to distinguish these two bus transactions in all of our simulations for all four protocols.

Cache block size	64 bytes
L1 size	64 Kb
L1 latency	3 cycles for data, 1 cycle for instruction
L2 size	2MB
L2 latency	10 cycles
Memory Latency	100 cycles
Memory bus width	16 bytes

TABLE I
SYSTEM CONFIGURATION PARAMETERS

Benchmark	Description	Input Parameters
ocean-noncontig	Ocean current simulation	258x258 grid
ocean-contig	Ocean current simulation	258x258 grid
raytrace	3-D scene ray tracing	teapot.env
lu-contig	Matrix factorization	256x256 matrix, B=16
cholesky	Cholesky factorization kernel	tk23.O
water-nsq	Forces in water	512 molecules
fmm	partical simulation	256 particles
radix	Radix sort	262144 keys
fft	six step FFT kernel	64K complex data points

TABLE II
SPLASH-2 APPLICATIONS AND INPUT PARAMETERS

C. Simulated System

We simulated a system with two procesors that were connected together via a shared bus. The specifications of each processor are shown in Table I. We plan to simulate with more than two processors in the near future.

D. SPLASH-2 Benchmark Suite

We used the SPLASH-2 benchmarks from Stanford in our simulations. The SPLASH suite was created specifically for parallel processor benchmarking. These programs represent a wide variety of computations in scientific, engineering, and graphics computing. The suite includes 8 full applications and 4 kernels. We were able to setup and run 9 out of the 12 benchmarks. The description of each benchmark and the input parameters we used in our experiments are shown in Table II.

V. RESULTS

The total number of Bus Invalidate transactions of all processors in the system is an important metric when comparing different invalidation-based protocols. Each Bus Invalidate transaction requires the processor to arbitrate for the bus and carry out the bus operation before it can proceed with the store instruction. Therefore, it is desirable to minimize the

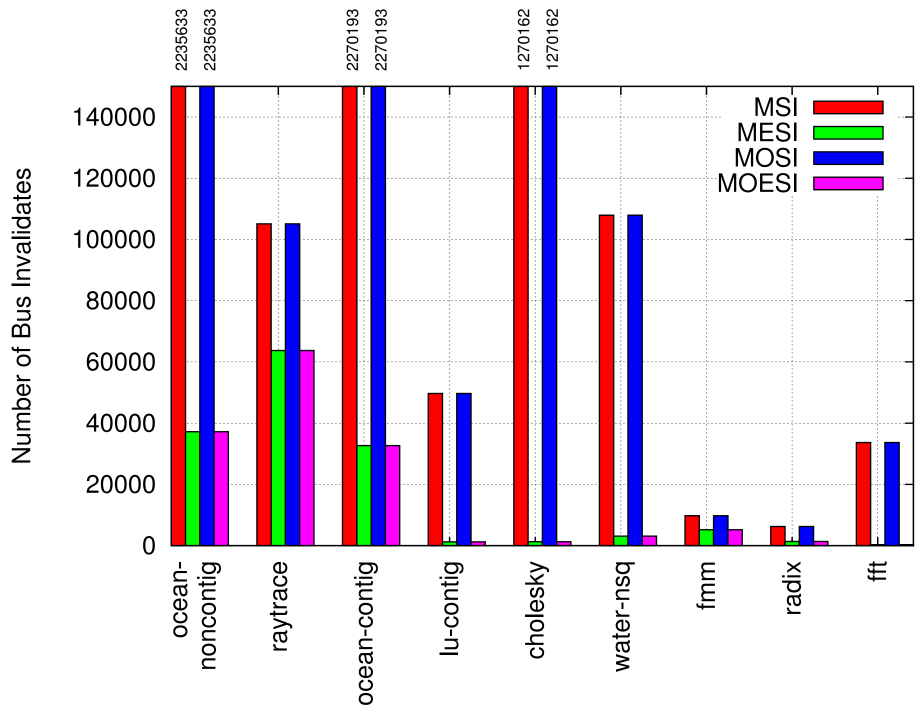

Fig. 2. Number of Bus Invalidates

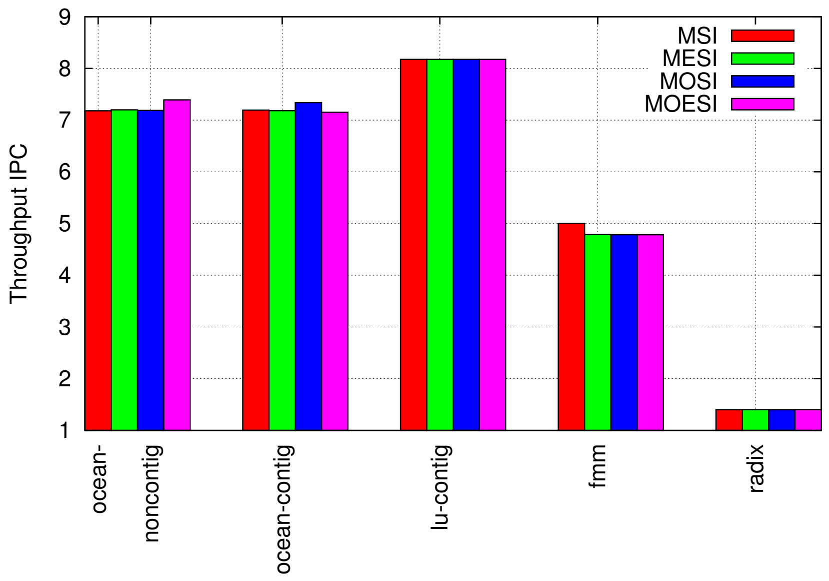

Fig. 3. Throughput IPC

number of times the processor has to send Bus Invalidations out on the bus.

We would expect those invalidation-based protocols that include an Exclusive state to have the least number of total Bus Invalidate transactions. Only those protocols with the Exclusive state can detect whether or not a cache block is *actually* present in another cache (shared). For example, on a read miss, in the MSI and MOSI coherence protocols, the requesting processor would always enter the Shared state even if the block was private. If this processor were to then write to that block, a Bus Invalidate transaction would have to be issued. On the other hand, in the MESI and MOESI coherence protocols, the requesting processor would enter the Exclusive state for the case of a private block. Now, when the processor wishes to write to that block, it can do so without any bus transaction. Since there are many references to private blocks, we would expect the MESI and MOESI coherence protocols to have significantly fewer total Bus Invalidate transactions when compared to the MSI and MOSI coherence protocols. In addition, we can expect there to be the exact same number of total Bus Invalidates between MESI and MOESI and also between MOSI and MSI. The reason for this is that both pairs of protocols handle references to private cache blocks the same way.

In Figure 2, we plot the total number of Bus Invalidate transactions of all processors in the system for each protocol across all benchmarks in the SPLASH suite. As expected, the total number of Bus Invalidates for the MESI and MOESI coherence protocols is significantly less than the number for the MSI and MOSI coherence protocols. In addition, the exact same number of invalidations occur for MESI as they do for MOESI. The same holds true between MSI and MOSI.

It is not appropriate to use only metrics directly related to cache coherence (such as the number of invalidations, write backs, etc.) to determine the performance of a cache coherence protocol. Other factors, such as processor lock-out, can have an offsetting impact on overall performance. Therefore, IPC is still one of the best metrics for comparing the performance of multiprocessor systems.

In Figure 3 we plot the IPC results we obtained by simulating each benchmark for the first ten million instructions. The y-axis shows the IPC throughput, which is the sum of the IPCs of all the processors in the system. As you can see, in the *ocean noncontiguous* benchmark, the MOESI protocol performs best, followed by the MESI protocol, which performs slightly better than both MOSI and MSI. However, for other benchmarks, such as *LU contiguous* and *radix*, not much difference in performance can be seen. Since we were not able to simulate the benchmarks in their entirety, the cold start effect could be severely affecting the performance numbers that we obtained. In addition, many of the SPLASH benchmarks first initialize their data structures in uniprocessor mode before spawning off multiple threads to parallelize the computation. This may also be affecting our results. We plan to run all benchmarks in their entirety in the near future.

VI. CONCLUSION

In multiprocessor systems where each processor has its own cache but memory is shared among all processors, there is a need to keep the caches *coherent*. In multiprocessor systems, such as an SMPs, where multiple processors and memory are all connected to the same bus, snoop-based cache coherence protocols are used to solve the cache coherence problem.

In this paper, we described several existing snoop-based coherence protocols. We also examined their deficiences, and discussed possible solutions to improve their performance. One such solution intended to improve invalidation-based protocols was Read Broadcasting, which allowed processors to snoop Bus Reads in the Invalid state with the hope of reducing the number of invalidation misses. We also discussed hybrid protocols, such as Competitive Snooping and the Archibald protocol, and the potential impact they can have on overall performance.

In the next section, we examined the hardware needed to implement snoop-based cache coherence protocols. Specifically, we outlined the major functions of the Bus Interface Unit (BIU). We also examined the various design issues associated with the BIUs of snoop-based multiprocessor systems.

In the final section, we presented our experimental methodology and the results we obtained from our simulations. We compared four cache coherence protocols (MSI, MESI, MOSI, and MOESI), using particular coherence related metrics (total number of Bus Invalidate transactions). The results we obtained matched our expectations, thereby validating our experimental methodology. We also presented the IPC results since it is one of the most important metrics in determining overall system performance.

REFERENCES

[1] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, 4(4):273-298, Nov. 1986.

[2] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt, "Network-Oriented Full-System Simulation using M5", *in Proceedings of the 6th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Feb 2003.

[3] D. Culler and J. P. Singh, Parallel Computer Architecture: A Hardware/Software Approach, San Francisco: Morgan Kaufmann Publishers, Inc., 1999.

[4] F. Dahlgren, "Boosting the Performance of Hybrid Snooping Cache Protocols", *in Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 60-69, June 1995.

[5] S. J. Eggers and R. H. Katz, "Evaluating the Performance for Four Snooping Cache Coherency Protocols", *in Proceedings of the 16th Annual International Symposium on Computer Architecture*, May 1989.

[6] D. Geist, A. Landver and B. Singer, "Formal Verification of a Processor's Bus Interface Unit", *IBM Case Study*, August 8, 1996. URL: http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/ps/biu.ps

[7] M. Loghi, M. Letis, L. Benini, and M. Poncino, "Exploring the Energy Efficiency of Cache Coherence Protocols in Single-Chip Multi-Processors", *in Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, pp. 276-281, April 2005.

[8] J. E. Veenstra and R. J. Fowler, "A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols", *in Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations", *in Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, 1995.