Coherence Lecture Notes

BRIAN RAILING

In taking a single processor system and replicating it to be a shared memory, multiprocessor, several challenges will arise. This chapter will focus on memory. How do processors reading and writing to memory ensure the appropriate values are used? And at its simplest, we can do away with caches so that all processors are always accessing memory. Correct, but an obvious performance loss. If those caches are restored to their processors and track accesses using tags, along with dirty and valid bits, is this sufficient to ensure correctness? No, and let's see an example.

Synchronization does not help with propigating the values, as the problem is the replicas of the data and not two operations using the value at the same time. Rather than just serializing the operations to a location in memory, we need to inform replicas that their values are stale and need to be fetched again from memory.

We claim that a memory system is coherent if it meets the following three properties:

- A read by processor P to address X that follows a write by P to address X, should return the value of the write by P.
- A read by processor P1 to address X that follows a write by processor P2 to X returns the written value... if the read and write are "sufficiently separated" in time.
- Writes to the same address are serialized: two writes to address X by any two processors are observed in the same order by all processors.

The first property maintains that the system obeys program order and operates as expected on a uniprocessor system. The second property is of write propagation, whereby writes must eventually be visible to the rest of the system. And the third property is of write serialization. Together these properties allow us to place the concurrent memory operations onto a single timeline and have all processors agree with this timeline.

Write serialization extends program order to apply to every processor's writes.

1 SNOOPING-BASED CACHE COHERENCE

Without caches, a multiprocessor system is coherent, all accesses are to memory. It is the processor caches that separate the processor from observing and updating memory, and for this separation we have to introduce additional logic (either software or hardware) to meet parallel program expectations that writes to memory are observed by the entire program.

There are several approaches to solving the problem of cache coherence, and the simplest is based on having each cache broadcast to the others when it starts executing specific actions. And then, the caches observe (i.e. snoop) these actions on a bus (or other on chip network) and may react appropriately in order to maintain the three required properties.

Unpublished working draft. Not for distribution

Unpublished working draft. Not for distribution

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2018 Copyright held by the owner/author(s).

1.1 Simple Writethrough Example

In exploring what makes a protocol coherent, we will start with an example design that uses only two states: V (valid) and I (invalid). This protocol will also be designed around the caches being write through, write no-allocate.

1.2 MSI

Beginning with a basic writeback, write-allocate cache, each line in the cache has two bits: valid and dirty. Each line can be in one of three states: invalid (valid bit not set), shared (valid bit set, dirty not set), and modified (valid bit set and dirty bit set). From these states, we can construct a set of coherence states by the same terms: M (modified), S (shared), and I (invalid). Each state must then obey certain properties within the larger system. A cache that is going to write (i.e. Modify) a line, must only do so when no other cache has a valid copy of the line. A cache that is going to read (i.e. Share) a line, must only do so when all other caches either have that line as invalid or are also sharing it.

For each state, we define a set of four actions that might cause transitions to different states. Two actions are initiating by the local processor: read and write. And two actions are observed from other processors: GetRd and GetRdX. Starting with lines in the invalid state, a processor that tries to read (or write) this invalid line will miss and send a request to memory. These requests are GetRd (for read misses) and GetRdX (for write misses). Each signals the intent of a cache to perform the associated operation on that cache line. It is important to note that a cache having a line in Shared state will still *miss* when the processor attempts to write to that line. These misses are coherence misses, operations that would hit in a uniprocessor cache, except for the maintaining coherence across the multiprocessor caches.

When a cache holds a valid line (either Shared or Modified) and snoops another cache's GetRdX action indicating the intent to modify the data in that line, the observing cache can no longer retain that line and instead must evict it for correctness, which requires writing back the data if the line was in the Modified state. During this time, the requesting cache is blocked, waiting on the state to be updated in other caches. And during which time, other caches may be completing read and write operations to their caches, provided that these operations require no bus operations. By this, the operations are not completing in the order they began execution; however, that is good as the system is executing concurrently.

As for evictions, this event is treated similarly to the uniprocessor model. In most cases, a processor evicting a cache line can do so silently, as a cache can always have less permissions than last granted by the system. Regardless, the key aspect of cache coherence is preserving writes, and so evictions do matter when the content of the line is dirty. In that case, the cache needs to gain access to the bus or other interconnect and send the block to memory. No permissions are exchanged, but the write to memory is observable.

Unpublished working draft.
Not for distribution.

1.3 Exclusive Problems

Excluding initialization, almost all writes are preceded by reads to that location. In MSI, the read would result in a GetRd and a transition from I to S. The subsequent write would require a second coherence request. However, if the reader was the first processor to recently access this data, then no other cache currently has the line when it requested read permission. So when it sends the second request, GetRdX, no other cache is invalidated. We can modify the protocol to optimize this sequence. When a GetRd request is issued, every cache checks whether the line is present, and if so, it signals on the bus (by driving a wire low or high). If no cache has indicated the line being present, then the requester has Exclusive (E) access. It is still a reader, but if it needs write permission, it will do so silently, as no other cache needs

to be notified. It can only do this *silent upgrade* while no other cache has accessed the line. If a GetRd request is snooped for a line in E state, then the line downgrades from E to S, as the cache no longer holds the line exclusively.

1.4 Cache to Cache Transfer

In introducing the exclusive coherence state, we required that all snooping caches report whether the line is present in their caches; however, the requesting cache still has the data supplied by memory. Memory which often takes significantly longer to access than other processor caches. One common extension is to then have one of the caches with the data present then supply the data.

This extensions leads to a new difficulty with having the caches determine which of them should supply the data. The simple case is when the data is known to be held by a single cache, being in either the M or E state. Or should just one cache hold the data in S, it can be the supplier. How should the hardware arbitrate when there are arbitrary number of caches holding the data.

The following are two solutions to this problem, neither calling upon the caches to arbitrate, but differing in how the situation arose. One approach is to add a new state, Owner (O), which is continually responsible for supplying the data to sharers. The Owner state is reached from the Modified state, when rather than writing back, the cache supplies the data to the requester. Thus the Owner still has responsibility of writing the data back to memory; however, should a processor write to the line, then the write back has been avoided. However, this scheme relies on the Owner continuing to supply data and not being evicted. This state is used in AMD processors.

The second approach is to add a new state, Forwarder (F), that is only responsible for making the next forwarding (or cache to cache transfer) of the associated data. After doing so, the sending cache downgrades to Sharer and the requesting cache first enters F. By keeping the most recent access of the data in the Forwarder state, this improves the chances that the next request will be serviced by a Forwarder. If the cache holding the data is Forwarder is evicted, the next request will be serviced by memory; however, following that request, a cache will again be holding the specific line as Forwarder. However, the Forwarder is otherwise identical to a Sharer and therefore the data is expected to be clean. This state is used in Intel processors.

> When a cache line has been invalidated, there is data still present in the block. One researched optimization is to use this data speculatively while waiting for the invalidating cache to supply the updated data. In many cases, the invalidation was either to a different part of the block or the prior value had been restored. If the data from the invalid block matches the supplied data, then execution has saved the time of the cache miss, otherwise, the processor will need to rollback its state to before the access (similar to a branch misprediction).

2 DIRECTORY-BASED COHERENCE

Snooping-based coherence implementations generally rely on the processor interconnection network being broadcast by default. If all memory operations require the use of the single common interconnection network (being broadcast), then this shared resource will eventually saturate as the number of processors using the network increases. While the specific saturation point varies depending on system configuration and workload, commonly by 16 processors there is a clear degradation in performance and minimal scaling benefits from increasing processor core counts. Even split transaction buses and other improvements can only do so much.

Unpublished working draft. Not for distribution.

At this point, the coherence protocol must no longer depend on broadcasting all memory traffic and instead limit observing operations to those of importance to the cache. When implemented properly, this works as if the caches still broadcast their operations, but some secret hardware filters out the notifications for which the caches are already in the invalid state. To do this, we must give some component the knowledge of where to find other caches that would need to observe the operation (i.e., have the target line in a state other than invalid). Let's start with placing this information in memory, as memory is already the place of last resort in coherence operations.

Continuing with cache lines as the unit of coherence tracking, memory now needs two pieces of information to track coherence. First, one presence bit per processor. Second, an additional bit indicating whether write permission has been requested. Assuming that the cache line is currently clean, any processor reading an address will send a request to the directory, which will update its presence bits with this newest access and supply it with the data. If the line is currently being modified, then the directory must forward the request to the cache holding the line in Modified state. This cache then write backs the data and downgrades either to S (if it was a read request) or I (if it was a write request). Again, the directory updates its state and presence bits. Finally, a write request that is issued when many caches are sharing the data requires all sharers to be invalidated. As the directory is tracking which caches are sharing the line, it can send invalidation requests to each cache.

Notice that the caches do not send any notifications to the directory when they evict lines in the shared state. Thus the directory's state may be stale. Should the directory send eviction messages to a cache that has already evicted, the cache can still acknowledge the request as that the cache line is already not present.

Unpublished working draft.
Not for distribution.

2.1 Space Reduction

Consider the space required by the directory, one bit per processor. At 512 processors, this is 64 bytes. 64 bytes is the common cache line size. Thus, memory would need 64 additional bytes to track the presence of each 64 byte unit of memory. While most systems are not so highly provisioned with processors, the presence bits still do consume a significant fraction of the system's memory and hardware must be provisioned for the worse case that it can support. Instead of using presence bits for every processor, the system could be designed to track the sharing of the data with fewer bits.

The first option is to design the directory with i entries, where each entry is a *pointer* (i.e. unique value) identifying the processor with a cached copy of the memory. Each *pointer* only uses $log_2 P$ bits, thus 512 processors requires 9 bits per entry. As processors read the line, each is added to the list of entries. If the number of entries required is more than the limit i, then the hardware must take one of several approaches to preserve coherence state.

Figure 1 shows how many processor caches had a line present, when a write forced them to invalidate. Over 99% of invalidates require notifying 6 or fewer caches. First, this reinforces that snooping is unnecessarily notifying other caches. Second, even with directories, tracking the presence of a cache line in a large percentage of available caches is unnecessary. While different workloads will vary in their level of sharing, the key is that most invalidations require only notifying a small subset of caches and therefore the directory only needs to track that degree of sharing.

2.1.1 *Dir-i-B.*

In the first approach, the directory has i entries and if more than the i entries are required, the system will revert to broadcasting (B) any change to this line. Since the set of processors sharing a cached line is a subset of all processors in the system, the directory can always broadcast any operations and the set of sharing processors will be notified. Other processors receiving the notification can ignore it, just as they would after silently evicting the line.

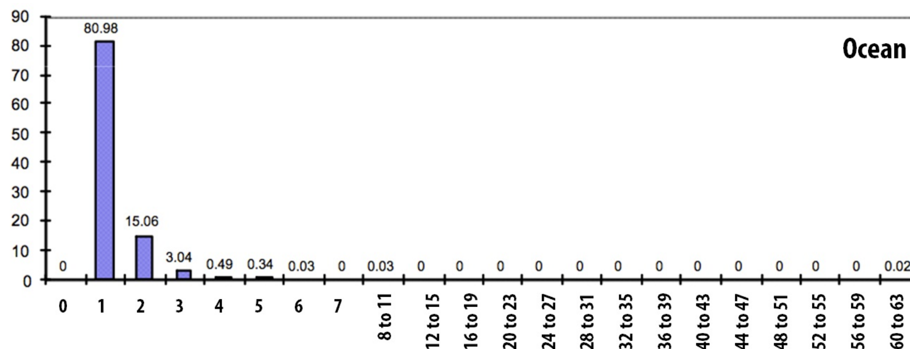

Fig. 1. Level of Sharing when a Cache line is invalidated by a write

Second, the decision to broadcast is on a per line basis. Other lines will have different sharing sets and therefore may operate with targeted notifications.

2.1.2 Dir-i-NB. In the second approach, the directory has i entries and if more than the i entries are required, the system will not broadcast (NB) and instead send an eviction / invalidation notification to one of the caches currently sharing the line. Given that caches may silently be evicting, there is some probability that the selected processor has already evicted the line. If the cache has not evicted the line, there is still a chance that the line would be evicted (either from replacement or required coherence) before its next access. Otherwise, the high degree of sharing has introduced an additional cache miss. However, the traffic generated by these occasional additional misses may be less than a broadcast.

2.1.3 Dir-coarse. In the third approach, each of the limited entries has several representations. By default, each entry is pointing to a specific processor. Should these entries be consumed, then the system reallocates the entries based on a different, coarser representation. Each coarser representation is based on treating increasing number of entry bits as *don't cares*. For example, entries for processor 6 ($0b110$) and 7 ($0b111$) can instead be stored as $0b11x$, where x is the don't care bit.

Coarser representations could also switch to having coarse presence bits, where each bit now represents sets of processors. For example, starting with 4 10-bit pointers (for a 1024 processor system), this provides 40 presence bits, so each bit can then indicate the presence within sets of 26 processors.

2.2 Distributed Directories

Rather than limit the number of entries in a directory, the directory can instead be designed to use space that scales with the number of processors, i.e. the cache. Memory maintains a single pointer to track one processor's cache that contains the data and can handle notifications. Each cache now has two pointers with its cache line, so that the set of sharers can be maintained as a linked list of caches. Thus any processor accessing the data has the space to track its sharing with respect to other caches, and this then scales with respect to the number of processors in the system.

On a read miss, the new requestor can insert at the head of this list. When a cache evicts a line, it needs to notify the other caches immediately before and after it in the sharer list. This step requires careful design as other caches could be evicting at the same time. On a write miss, every sharer must be invalidated, which takes $O(sharer)$ time, rather than the constant time of most other operations.

Unpublished working draft
Not for distribution

2.3 Intel

Intel L3 caches maintain a directory for the lines contained within the L2s. Thus the memory only needs to track on a processor package level, thus scaling down significantly the bits required for tracking.

2.4 Operation Reduction

The basic implementation required 5 network transactions to service a read miss, when the content is dirty. The operations are as follows:

(1) P0 misses and sends request to directory
(2) Directory returns ID of processor (P2) with dirty data
(3) P0 sends request to P2
(4) P2 returns data to P0
(5) P2 sends data to home directory

Each operation is in the critical path, except for the last. We can reduce the traffic and path in two ways. First, the directory can send the invalidate and forward request directly to P2 rather than back to the requestor, P0. Then P2 sends the data back to the directory, which forwards it to P0. This avoids one transaction, but the critical path is still four.

The critical path can be improved by changing the response. The processor instead send two transactions, as per the original list, one to the directory and the second to the original requestor. This reduces the path to three, but adds complexity in that the system no longer follows a request / response pairing.

Unpublished working draft
Not for distribution

3 MEASUREMENTS OF MEMORY SHARING AND COHERENCE

4 IMPLEMENTING COHERENCE

4.1 Inclusive Problems

In a uniprocessor world, having multiple caches in the memory hierarchy can provide more total space. Each cache can independently store data, such that loading a line from the L2 into the L1 cache can be instead treated as a swap with the evicted line taking the place in the L2 cache. This increases the total capacity of the processor's caches, as blocks are not duplicated in different levels of the hierarchy. This design is termed exclusive caches.

When there are instead multiple processors in the system, an exclusive cache can pose a difficulty. Actions can come from the interconnection network that cause evictions or downgrades of lines in a processor's cache. But with each cache holding copies of the data exclusive of the others, each action received from the network must be reviewed by every cache. In contrast, were the cache to be inclusive, then the last level would be capable of making any determination regarding whether the other, smaller caches also hold copies of the data just by consulting its own status. Reducing the need for the network actions to have to access the smallest caches helps those caches to perform faster.

Furthremore, each cache relies on LRU or some variation thereof to determine the next line to evict. As the access stream only reaches the cache level that can service the request, higher levels of the cache do not update the LRU information in the same ordering. We can help address this issue by having the L1 inform the L2 on each of its internal evictions.

The final inclusive problem comes from the knowing which cache has the latest modified copy of the data. When another cache forces the invalidation of a modified line, the cache needs to write its contents back to memory. The L2

may not have the latest write, even if it has the line as modified / dirty. The cache can then furthermore maintain a bit indicating whethe the L1 will need to write its contents back to the L2, or if the L2 can supply its block.

5 PROBLEMS

(1) Starting from MSI, what state, if any, would you add to a processor's coherence protocol if it was often used in a uniprocessor system? Why?

(2) How many limited pointers could fit in the presence bits required for 256 processors? How much overhead does this design impose if there are 64 byte cachelines? What is the overhead as the pointers scale from 1 to the maximum computed before?

Unpublished working draft
Not for distribution