An Analysis of Directory-Based Cache-Coherence Protocols on Symmetric Multiprocessors Using the SESC: SuperESCalar Simulator

Eric Chang

Abstract

The directory-based protocol is based on something. This report will discuss the advantages and disadvantages of using one type of protocol vs another type. I will introduce MSI and MOESI protocols, then a basic directory, followed by a protocol based on the implementation found in the SGI Origin server computer. I will then go into detail about the specific simulator implementations. Finally, results based on the simulations are presented.

1. Introduction

Computers today are moving increasingly towards multi-core architectures because we have reached a thermal barrier in increasing transistor switching speeds. As such, it is important to figure out ways to implement effective multi-core architectures and how schemes offer different tradeoffs between speed and memory consistency. In particular, the cache-coherence schemes used by multi-core processors have a huge impact on the performance of a multi-core architecture. This report will focus on various cache-coherence protocols and mechanisms for carrying out those protocols.

Using mechanisms like a snoopy bus or a directory, we enable the use of cache-coherence protocols. Since directories and buses are basically ways for multiple caches to communicate with each other, it is important to understand also what the cache protocol being used is and what kind of requests they can send to each directory. "In a coherent multiprocessor, the caches provide both migration and replication of shared data items" [1]. It is important for the architect of the processor to design these features into the processor as to allow the programmer to take advantage of the speedup available in having multiple data.

2. Architecture

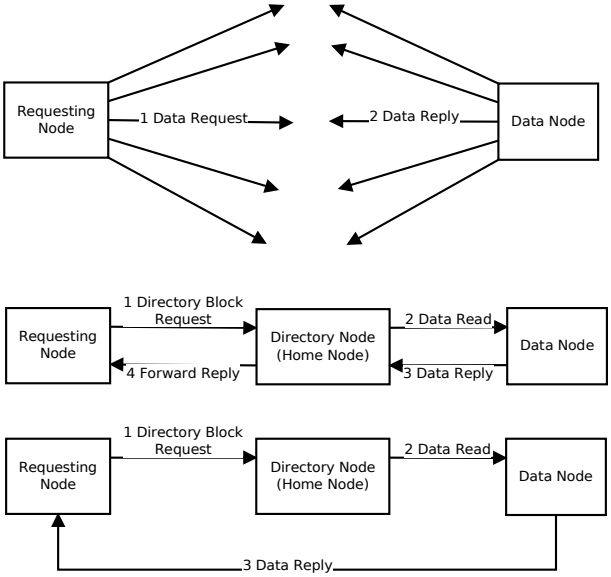

Figure 2.1 Top: snoopy-based cache-coherence protocol. Middle: regular directory-based cache-coherence protocol. Bottom: SGI Origin cache-coherence protocol.

There are a couple of different types of cache coherence protocols and each of them provide their own advantages and disadvantages. In this report, I will be focusing directory-based protocols. While taking care of cache coherency, we also need to keep in mind that Amdahl's law prevents us from improving our speedup to a certain degree if our program contains sequential code. Since sequential code cannot be sped up, we can never achieve a speedup faster than the time it takes to run that sequential code.

2.1. Mechanisms

The cache-coherence protocols are responsible for maintaining data coherency, but it is necessary for it to run on top of mechanisms. Mechanisms for cache-coherence protocol include snoopy-based and directory-based protocols.

2.1.1. Snoopy-Based Cache-Coherence Protocol

The snoopy-based cache-coherence protocol relies on the bus to transfer necessary information. All messages are broadcast on a common bus, which is monitored by each processor and its cache. Each processor needs a controller to snoop the bus. When encountering an important message that it needs to act on, it changes the cache. When the processor requires something, it sends its request on the bus. The broadcast messages on the bus is heard by every processor.

Snoopy-based protocols have an advantage when it comes to manufacturing because they can use the existing bus to memory as the broadcast medium for communicating information about cache coherence [1]. However, a snoopy-based protocol is not as scalable as a directory-based protocol. This cache-coherence protocol will not be presented in the final results because this report is focused on comparing differences in directory-based cache-coherence protocols.

2.1.2. Directory-Based Cache-Coherence Protocol

Directory-based cache-coherence protocols are a class of widely used cache-coherence protocols that has been proven in the past to be able to scale up compared to cache-coherence protocols based on the snoopy method. "As processor speeds and the number of cores per processor increase, more designers are likely to opt for [directory-based cache coherence] protocols to avoid the broadcast limit of a snoopy protocol [1]. Directory-based protocols are interesting because it can achieve higher speeds than snoopy-based protocols. A snoopy-based protocol achieves very high consistency because all messages are broadcast on a bus, meaning all processors know whatever changes are happening in the system. However, in a directory-based protocol, it is possible for the directory to be in an inconsistent state for a longer time. For example, when we are evicting a block from a processor, this is broadcast across the bus in a snoopy-protocol, and the directory knows right away to change the directory state. In a directory-based cache-coherence protocol, there is some time between when the eviction message is sent from the owner to when the message arrives at the directory. In this time, it is possible for some other request to come in to the directory, allowing the directory to possibly fetch the invalidated data from the owner that just evicted its data.

A directory-based cache-coherence protocol can be designed several ways. The directory needs to keep track of which CPU has which cache line. This is necessary because since the directory acts as the communication between each CPU. In a snoopy system, we have no way of keeping track of which CPU has which cache block. This is the advantage in the directory-based cache-coherence protocol that allows fewer messages to be sent. One of the easiest ways to keep track of the directories is to keep a bit vector in each distributed directory about which CPU has which cache block. Another way would be to keep track of the node ID in the directory. The advantage of using a node ID is that it can potentially take up less space in the directory when we have a large number of processors, since we do not have to keep track of all processors. However, there is a disadvantage to keeping track of the node ID Usually, in a node ID based directory-based cache-coherence system, we do not keep track of all the node IDs because that would defeat the primary purpose of using the node IDs, which is to save space. Therefore, when

we have a system that keeps track of the cache block using node ID, we are unable to keep track of all the CPU's that might potentially request for a cache block. In that case, we would have to invalidate a CPU when we want more space in the directory. Using bit vector to keep track of the CPU that has a specific cache block is an efficient way to implement directory-based cache-coherence protocol, but it has a certain limit on how many CPU's can be used together with the system.

The directory is required to store more than just which CPU currently contains a cache line from the current directory. It is also necessary for the directory to store what state the directory is in in order to find out which processors own or share a block. If we did not store the state of the directory, we would have to query all the processors to find out what state they are all storing, which would be incredibly inefficient. The following are the possible data that one can hold in each cache line in the Directory:

M : {P}
S : {any number of P}
E : {P}
I : {NULL}
O : {P-owner} {any number of P}

In the bit-vector case, we can have an extra entry to store the address of the owner. Or we can indicate owner using an extra bit in each cache line. We can also keep owner at the top of the list, to indicate an owner. Keeping track of the owner could be done by either adding a bit to each directory entry, or making sure the owner is on the very top, or simply not keeping track of it at all and using S for it. In the last method, the node that contains the owner would be responsible for updating other directories that are requesting for the value. However, this would be somewhat difficult because we cannot know which node holds the modified copy. In a snoopy protocol, this is not a problem, since the owner can just intercept a read request. However, in a directory-based cache-coherence protocol, the owner directory cannot see all the requests. Therefore, the directory should keep track of the owner somehow.

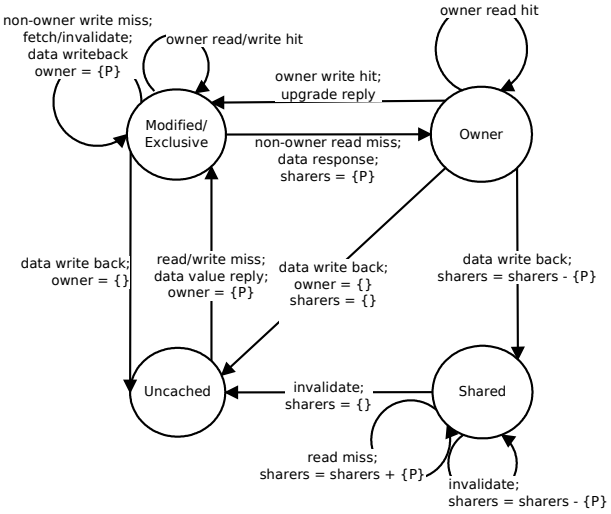

Figure 2.2 State diagram for directory-based protocol

In any directory-based cache-coherence protocol, there are three logical nodes in any request: the requesting node, the directory node, and the data node. Of course, these logical nodes can all be the same physical nodes or they can all be different, but it is easier to think of them logically as three separate nodes. The Requesting node is the original requester, the node that sends out the original read or write request. Since this is a directory-based protocol, it is necessary to send the request to the directory in order to find out where the data actually is, what state the directory block is in, and whether or not the request can be satisfied immediately.

The basic directory protocol is easier to implement than some of the more advanced and newer version of the protocol, but it is also unoptimized. This protocol makes sure that each request has a corresponding reply and that no operations can proceed until the responding message has been received. No assumptions are made when deciding the

directory state because we wait for the owner or shared block to return to the directory before we forward it back to the requester.

In the simulator, we make sure that whenever a message is being sent to a remote node that has the same node ID as the current node, we send the message to itself. For example, if processor 1 requests a block A, where processor 1 is block A's home node, sending a network message from a node to itself it pointless and just results in needless network traffic. We perform a check to see if the destination for a message is the same as the source of the message, previous to sending the network message. If this is the case, it just forwards it straight on to the function that would handle it on the local directory node rather than emitting something on the network.

2.1.3. Modifications to the Directory Protocol

There is an optimized version of the basic directory protocol mentioned in the previous section. The SGI Origin implemented this protocol, which is based on an altered version of the protocol used in the Stanford DASH multiprocessor. This protocol tries to be more optimized by eliminating unnecessary communications between the data node and the directory node. It achieves this by assuming that all read requests to the directory can be satisfied, and if it cannot, it is up to the owner of the data to send an additional invalidate back to the directory as well as the data response to the original requesting node. We save bandwidth by allowing the data packet retrieved to go directly to the requester instead of going to the directory first.

This type of protocol can introduce many opportunities for deadlock. As a result, the system sometimes need to send out two messages simultaneously, as opposed to the basic directory, where all operations can only cause one outgoing message to be sent on every incoming message.

The protocol deals with this deadlock in various ways. The first method is to have the directory change to Busy-Shared or Busy-Exclusive whenever the directory state and the request cannot be satisfied immediately. This situation can happen, for example, when the directory state is Exclusive with another owner, and a read request comes in. This ensures that the directory stays in a consistent state and is not modified based on invalid directory state.

2.2. Cache

Since directories and buses are basically ways for multiple caches to communicate with each other, it is important to understand what caches are and what kind of requests they can send to each directory. "In a coherent multiprocessor, the caches provide both migration and replication of shared data items" [1]. It is important for the architect of the processor to design these features into the processor as to allow the programmer to take advantage of the speedup available in having multiple data. Having multiple data allows for multiple reads at the same time. The two protocols introduced here serves as an example of some of the protocols that can be used to achieve cache coherency. The MSI protocol is the most basic one and only uses three stages. The MESI protocol adds an "Exclusive" state to reduce the traffic caused by writes of blocks that only exist in one cache. The MOSI protocol adds an "Owned" state to reduce the traffic caused by write-backs of blocks that are read by other caches. The MOESI protocol implements both the Exclusive and Owned state to take on both characteristics.

2.2.1. MSI Protocol

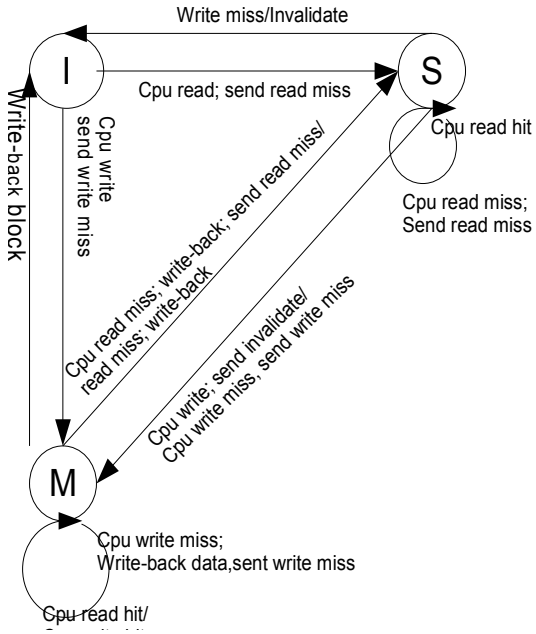

Figure 2.3 MSI cache

The MSI protocol is the most basic of the cache-coherence protocols, using only three stages to ensure cache coherency. The three states stand for Modified(M), Shared(S), and Invalid(I). The Modified state signifies that the cache block is modified and that no other cache contains the entry. This state is necessary whenever the processor needs to perform a write operation. The Shared state means that the cache block can exist in caches other than the current one, and Invalid means that there is nothing usable in this particular cache block.

Because this protocol contains only three states, it saves space on storage as compared to some more elaborate schemes. The disadvantage is that this protocol requires more messages on average to be sent in order to achieve the same level of coherency. It is easy to see why, because adding the Exclusive state allows the cache to not send a message when downgrading from Exclusive to Shared. In the MSI protocol, there is no way to distinguish between a dirty exclusive and a clean exclusive, meaning there are situations where we will write a clean block to memory. Only having three states does simplify the implementation.

2.2.2. MOESI Protocol

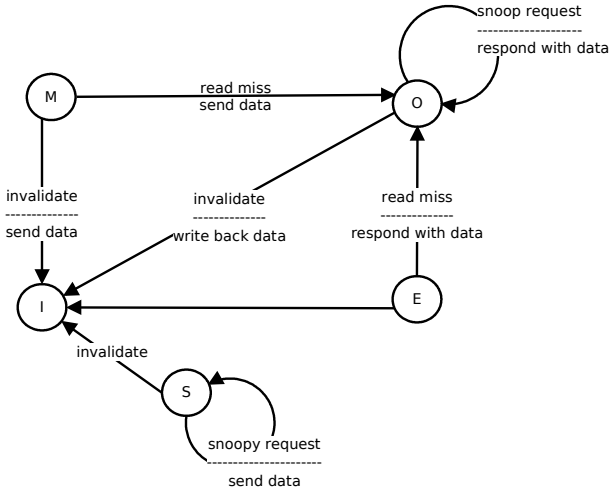

Figure 2.4 MOESI cache, requests from directory

The MOESI protocol is an improved protocol upon the MSI protocol in that it adds both an Owned(O) state and an Exclusive(E) state to the protocol. The Exclusive state is advantageous in that it reduces the traffic caused by writes of blocks that only exist in one cache. The inclusion of the Owned state is beneficial in that it reduces the traffic caused by write-backs of blocks that are read by other caches. In other words, it allows dirty lines to be shared quickly. These two added states allow the protocol to send fewer messages and achieve a lower latency in certain operations. A disadvantage of this protocol is that it uses the most amount of space to store this information, since it needs to store more states, compared to MESI, MOSI, or MSI protocols.

2.2.3. Changing Cache Size

	Main Memory	Cache Line	Other Processor	Snoop Request
Modified	Stale	Most recent, correct copy	Hold no copy	Modified cache responds
Owner	Can be stale	Most recent, correct copy	Shared state	Owned cache responds
Exclusive	Most recent, correct copy	Most recent, correct copy	Hold no copy	Exclusive cache or main memory responds
Shared	Can hold most recent, correct copy	Can hold most recent, correct copy	Shared or owned	May not respond
Invalid	May hold valid or invalid	Invalid copy	May hold valid or invalid copy	May not respond

Table 1 Acceptable states for MOESI cache

The cache size can be determined by associativity * number of sets * width. Therefore, in the configuration file, it is only necessary to give these three parameters and not the total size of the cache. For example, in my system, this could be 4 * 4 * 64, in the case of the L1 cache. This gives it 16 blocks of 64-bit data to have a 1kB L1 cache. In the case of the L2 cache, I used an associativity of 4 with 8 sets to get 32 blocks. this equates to a 2kB cache.

Increasing cache size increases coherence misses because more invalidates occur because fewer blocks are bumped due to capacity misses. Of course, capacity misses decrease because the cache has more spaces to put blocks [1]. Increasing block size means capacity miss decreases and compulsory miss decreases for certain applications. When this happens, it most likely means that there is a lot of spatial locality in the code, such as when running kernel code. Because increasing block size grabs more of the code in the same area together, which directly reduces compulsory misses. The capacity miss is reduced because we're storing more of the necessary code in the cache [1].

3. Implementation

The implementation of the code was done in C++, using the SESC Simulator as an underlying layer. It supports various features of a computer system that allows us to simulate the differences between different directory-based protocols.

3.1. Network

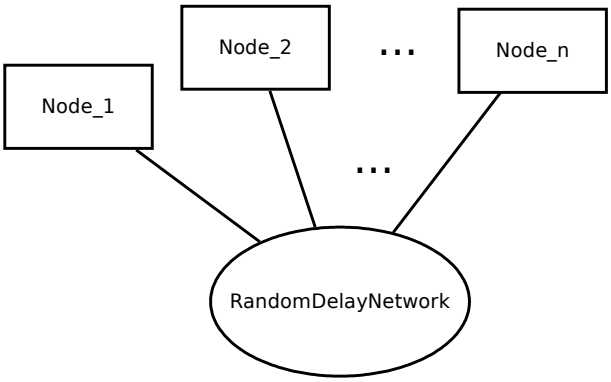

Figure 3.1 Node connections to the network

The underlying network in use for this simulator is a simple black box model. It does not model a real network with routing. Instead it models messages going in and out of the network using a random delay with a lower-bound of 4 and an upper-bound of 20. When more messages arrive, the delay coming from the random delay generator will be shifted higher as to model the higher traffic conditions.

We could have used a more complicated network, one that simulates router-router connection and uses routing protocols, but such a network was not available at the time in the simulator, and the black box network

serves its purpose for delivering messages to and from all the directories, as well as the main memory. Although the SESC simulator has the capability to model a more complicated network, a simple network, such as the one used here, can illustrate our point.

3.2. Main Memory

In the simulator, the main memory is simulated as a node in the network. Therefore, when a directory wants to request something from main memory, it sends a messages across the network to the main memory. This design simplifies the design of the memory while still keeping the protocol intact.

3.3. SESC Simulator

The connections in the SESC Simulator are shown in Figure 3.2. We are using an architecture with a 1K L1 cache and a 2K L2 cache.

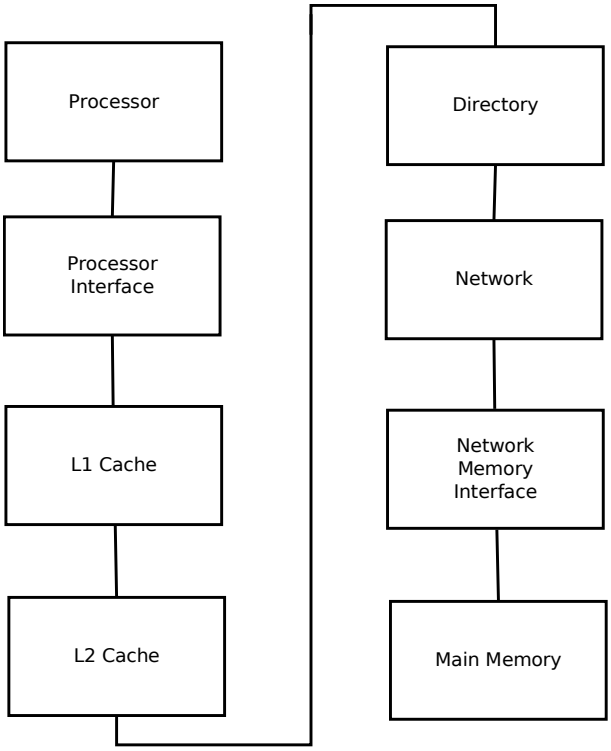

Figure 3.2 Connections in a single node

4. Results

The results are shown in Table 2 for the results using the benchmark fft in the SPLASH-2 benchmark suite with 32 processors. The fft benchmark is a complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm, which is optimized to minimize inter-processor communication [2].

4.1. Verifying Correctness

To verify whether or not the simulator is correct, we run the benchmark on a normal machine to find out what the output is, then run the program on SESC. The output produced from SESC should be identical to that produced by running the benchmark on a real processor. If not, then it means that the program that simulates the directory protocol is not running correctly. In addition, the SPLASH2 benchmarks' kernel programs provide self-test that we can invoke to ensure that our protocols were implemented directly. It achieves this self-test using inherent tests to the data structure.

4.2. SPLASH2 benchmarks

Using these results, we can see that the MSI protocol takes longer to complete. Although there are more read misses for the MOESI cache, it is offset by the amount of read hits for L2 cache. It could also be possible that the additional read misses do not incur as much penalty in a MOESI cache because in a MOESI cache, there is a greater opportunity for the cache to fetch the data from another node in that directory instead of from the memory.

We also see that for both MOESI and MSI caches, as the CPU count increases, the run-time decreases, this is to be expected as more work can be done in the same amount of time when one has more processors that can be used at the same time.

Benchmark	Runtime	L1Cache Read Hit	L1Cache Read Miss	L1Cache Write Hit	L1Cache Write Miss	L2Cache Read Hit	L2Cache Read Miss
MOESI 2	1123774	104129	6225	72332	7978	14203	12478
MSI 2	1125149	104129	6221	72332	7977	14198	12479
MOESI 4	498957	104907	6332	72760	5226	11559	9742
MSI 4	497389	104907	6315	72763	5226	11545	9732
MOESI 8	286050	106467	6521	73628	3852	10373	8030
MSI 8	288279	106467	6513	73628	3871	10384	8052
MOESI 16	174171	109589	8079	75361	3035	11128	8352
MSI 16	175160	109589	8094	75361	3032	11142	8339
MOESI 32	138960	115825	10693	78816	3179	13904	11562
MSI 32	142046	115825	10670	78816	3173	13880	11256

Table 2 Results for fft

5. Problems

In implementing the directory-based cache-coherence protocol, there were some problems. One was simply that debugging such a large system is inherently hard, especially since the bugs often surface after tens of thousands of messages are sent. It is useful to print out each message that pass around the system. To debug these problems, it is useful to know which messages cause which transitions. For example, if we see that a message is being sent to the directory node when it should be sent to the requesting node, we know that there is a problem. The debugging system works by various lines of assertions that should always hold true if the system is working. They can easily be turned off to increase speed of the simulator by not defining DEBUG when compiling.

6. Further Work

The memory system in the current system is implemented in a way such that it is in one location. In the future, we could modify the the simulator such that the memory system is distributed just like the directory system. With a distributed memory system, there would be much more issue with coherence, scaling, performance, and correctness.

We can compare the protocol we have to a snarfing protocol. In this protocol, the cache controller watches both address and data in an attempt to update its own copy of a memory location when a second master modifies a location in main memory. When a write operation is observed to a location that a cache has a copy of, the cache controller updates its own copy of the snarfed memory location with the new data.[3][4]

References

[1] John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 2007.

[2] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 24-36, June 1995.

[3] Keith I. Farkas and Norman P. Jouppi, Complexity/Performance Tradeoffs with Non-Blocking Loads. In Proceedings of the 21st International Symposium on Computer Architecture, pages 211-222, April 1994.

[4] James Laudon and Daniel Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server. In Proceedings of the 24th International Symposium on Computer Architecture, pages 241-251, May 1997.