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

Eric Chang

May 18, 2011

Abstract

Directory-based cache coherence protocols are based on the central concept of having the states of any particular cache block located in a known, fixed, location. This report will discuss the advantages and disadvantages of two different directory-based cache coherence protocols. I will first give an overview of all the components in the system. Next, I will introduce select components and message types in more detail. Then, I will introduce various cache coherence protocols in general, as well as the specific protocols that will be compared with each other, which are a directory-based cache coherence protocol based on bilateral interactions and a different protocol based on the one used in the SGI Origin 2000-based systems. I will then go into detail about simulating and implementing the differences between the protocols using the SESC: SuperESCalar Simulator. In addition, results from simulations using SPLASH-2 benchmarks are presented. We show that the Origin directory-based cache coherence protocol has several advantages over the directory-based cache coherence protocol based on bilateral interactions.

1 Introduction

Computers today are moving increasingly towards multiprocessing 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 multiprocessing architectures and how schemes offer different tradeoffs between speed and memory consistency. In particular, the different schemes used for cache coherence by processors have a noticeable impact on the performance of a multiprocessing architecture. Cache coherence protocols require mechanisms like snoopy buses or directories to function. Therefore, it is also important to understand how underlying mechanisms such as buses and directories allow multiple caches to communicate with each other.

For this purpose, I will present the cache coherence protocol shown in “The SGI Origin: A ccNUMA Highly Scalable Server”[1]. In this report, I will go in depth into the Origin system to show how the cache coherence protocol in this system operates. For comparison, I will also present a detailed analysis about an unoptimized protocol based on bilateral interactions, or Bilateral Interaction Protocol (BIP). Cache coherence protocols based the Origin protocol and BIP are both programmed and simulated in the SESC in order to find out which one is faster, and why it is faster. However, one thing to note is that the Origin 2000 system mentioned in the SGI Origin paper is not directly applicable to the CMPs (chip multiprocessors) of today because it was designed for older systems. In particular, this system employs distributed shared memory (DSM), in which part of the main memory with every node[1]. More details will be provided in the next section about this situation.

To show the differences between these two protocols, I show their performances according to several benchmarks. The Origin protocol is superior than BIP because BIP uses more messages than the Origin protocol for several different types of requests. To demonstrate this, I created a synthetic benchmark that generates many requests, which will generate different amount of messages between Origin protocol and BIP. I will also show the protocols’ performances on real-world benchmarks. This will demonstrate their performance when faced with benchmarks that are not specifically geared to exploit the differences between

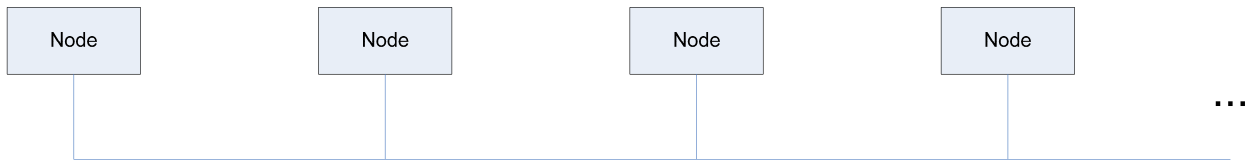


Figure 1: Connections in the system

the protocols. The experiments I ran will indicate that the difference in protocol is what causes the difference in performance, and not because of some other differences in the simulator.

In the following section of the paper, an overview of how each components are connected together to form the complete system is presented. Section 3 shows the details of each individual component in the system. Section 4 talks about the basic message types that are present in the system. Implementation and comparison of protocols is presented in Section 5. Section 6 discusses the experimental setup used in the simulation. Performance and results of the two types of protocols is presented in Section 7. Section 8 describes some of the problems faced in simulating the protocols and some possible future works. Finally, section 9 concludes the report.

2 Overview of System

A block diagram of the system is shown in Figure 1. This is the system that will be used to compare the two protocols. From the figure, we can see that a network connects identical nodes together to form the system. The basic building block of this system is the node, which is shown in Figure 2. This system is composed of the hub, the directory, the memory, the L2 and L1 cache, and the processor. The hub is responsible for directing incoming network messages to the directory or to the L2 cache. The hub is also responsible for any messages that might be sent between the directory and the cache of a single node. When the directory sends a message to a local cache, or vice versa, the hub will redirect the message to the receiver without sending this message on the network. The L2 cache is connected to the L1 cache, and the L1 cache to the processor, to form the complete node.

This system is a close approximation of the Origin system, and it will be used to run the simulations in the comparison of protocols[1]. This system uses a DSM, and each memory is addressable from any node. If any request arrives at the directory where it is necessary to retrieve the memory, the directory is guaranteed to be attached to that memory and does not have to ask any other node, since each directory is responsible for accessing its own memory. This system is not similar to today's systems because it uses a DSM, where memory is distributed across each node. This is not the case on a modern CMP. However, if we consider a CMP where the last-level cache is distributed across the nodes (cores) and ignore misses from the last-level cache, then the interactions of the protocol becomes similar to what can happen in a modern CMP system. The individual components in this system will be introduced in more detail in the next section.

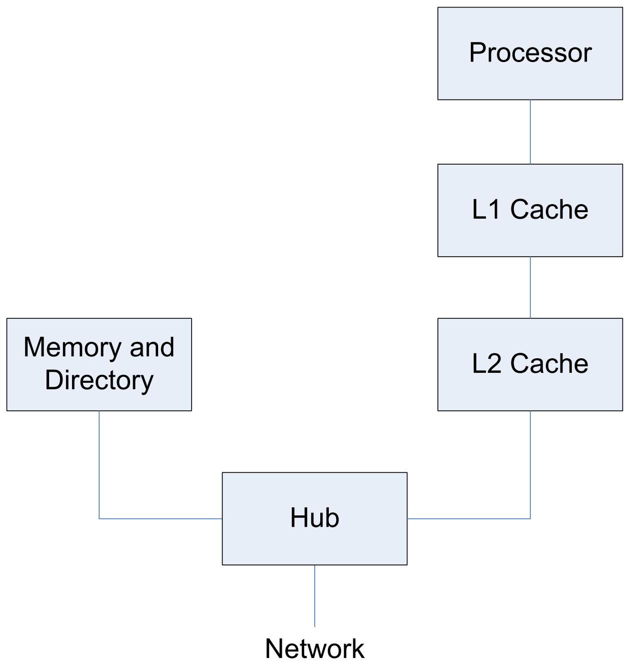


Figure 2: Inside the Node

3 Components of System

The most important job of any cache coherence protocols is to maintain data coherency. In order to maintain data coherency, the protocols all need to make sure that whenever a cache writes to its block, it has exclusive access to it. Because of that, any cache coherence protocol involves maintaining states for the blocks in the caches. Each individual cache needs to know the state of its own block, but the caches also need to communicate to each other about block states by sending messages to each other over a network. In a directory-based cache coherence protocol, the directory is responsible for storing some of these states. I will introduce the network that connects the individual nodes together, followed by the cache and the directory.

3.1 Network

The network connects the different nodes of the system together. It is responsible for making sure that the various nodes receive the messages correctly and allows the nodes to share resources via internodal communication. It is important that the network does not slow down the system or cost too much to build. Since many messages must travel through the network as part of the communication between different nodes, the network needs to be designed to transfer as much information as possible in the shortest amount of time without incurring a substantial cost.

There are several different ways to deliver a message in a network, which are broadcasting, multicasting, and unicasting. Broadcasting refers to sending out the message to all nodes connected to the network; multicasting refers to having some mechanism in the network to create copies of messages for sending from a single source; unicasting means the transmission of each message is intended for only one device. The system that is used here uses unicasting, because it sends out individual messages into the network for every single request or response.

There are some latencies involved in sending messages into the network. First, there is the amount of time it takes for sending or receiving node to process the messages before putting it in the network. In addition to the sending overhead and the receiving overhead, there is the network latency. This is how long the messages spends in the network. Essentially, it is from when the first bit of the packet arrives into the network to when the last bit leaves the network. The overhead at the sender and the receiver is not included in this latency. In most cases, these overheads are shorter than the network latency, so it is possible to saturate the network if messages are sent continuously. However, that should only happen if the workload is tuned to generate lots of messages into the network[1][2].

3.2 Cache

Since the cache is an integral in a coherence protocol, it is important to understand what caches are and what kind of requests they can send to each other and to the directory. "In a coherent multiprocessor, the caches provide both migration and replication of shared data items"[2]. It is important for the architect of the processor to design these features into the processor to take advantage of the speedup available in having multiple blocks of the same data across different nodes, while still giving the correct results. There are several states that has to be stored by the cache in order to enable the rest of the system to function properly.

The two protocols introduced here serves as an example of some of the protocols that can be used to achieve cache coherency. I will go over the states stored in the MSI cache protocol first. This protocol allows the least amount of information to be stored, yet it still provides enough information for a cache to acquire exclusive access. An improvement over this protocol can be found in the MESI protocol, which adds an "Exclusive" state to reduce the amount of unnecessary operations in the system.

3.2.1 MSI Cache

The MSI protocol is the most basic of the cache coherence protocols, using only three states to ensure cache coherency. The three letters stand for Modified, Shared, and Invalid. The Modified state signifies that the cache block is in a dirty exclusive state and that no other cache contains the entry. This 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 are no usable data in this particular cache line.

Because this protocol contains only three states, it can save space on storage as compared to some more advanced schemes. In addition, the protocol for a cache coherence protocol designed using less cache states can be simpler. The disadvantage is that this protocol requires more messages and higher latency on average as compared to protocols where the cache utilizes more states[2].

3.2.2 MESI Cache

	Main Memory	Cache Line	Other Processor
Modified	Stale	Most recent, correct copy	Hold no copy
Exclusive	Most recent, correct copy	Most recent, correct copy	Hold no copy
Shared	Can hold most recent, correct copy	Can hold most recent, correct copy	Shared
Invalid	May hold valid or invalid	Invalid copy	May hold valid or invalid copy

Table 1: States in MESI cache

The MESI protocol is an improvement upon the MSI protocol in that it adds an Exclusive(E) state to the protocol. In a system that uses the MESI cache, Table 1 illustrates all the states that other components can be in when we have knowledge of what the current cache holds. The Modified state means that only the cache that holds the block has the correct copy, and it has to supply that block to certain requests. The Exclusive state means that the main memory and the cache both have the same correct copy, so either can respond, depending on the protocol. When the block is Shared, all caches that has the block and the main memory has the most updated copy. When the block is Invalid, the current cache does not know anything about the block and does not own the block. Whenever a cache holds a block in the Exclusive state, it means that the cache has Exclusive access to the block, but had no intention of altering the block at the time of request. In other words, the Exclusive state indicates a clean exclusive state. Adding the Exclusive state is advantageous in that it has the potential to reduce traffic when it comes to writes and exclusive read operations. A disadvantage of this protocol is that it requires more space to store this additional information as compared to the MSI protocol.

Once we add this state into the protocol, certain cases that required an additional request into the network in the MSI protocol becomes an operation where no additional messages are emitted. For example, the MSI protocol could fill an Exclusive reply as Shared, and it would become necessary to send an additional network message to request for exclusive ownership when transitioning from Shared to Modified. In a MESI cache, since it has the Exclusive state, the same transition would go from Exclusive to Modified without emitting any messages on the network. If instead, the MSI protocol fills an Exclusive reply as Modified, there would be situations where we would write a clean block to memory. With an additional Exclusive state, the cache does not have to write back its block to memory when downgrading from Exclusive to Shared. Although directory-based cache coherence protocols can use the MSI cache, the MESI cache is the cache that will be used in both of the directory-based cache coherence protocols presented in this report.

3.3 Directory

The directory is responsible for tracking the state of each cache block in order to implement these two operations: “handling a read miss and handling a write to a shared, clean cache block”[2]. This is necessary since the directory acts as the communication channel between each CPU. In a snoopy system, we do not keep track of which CPU has which cache block in a centralized location. This is the advantage in the directory-based cache coherence protocol that allows fewer messages to be sent. Therefore, it needs to have a list of nodes that have the cache block. If we did not store these in 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.

Cache State	Directory Data Structure
Modified (Dirty Exclusive)	Owner = {P}, Sharers = {}
Exclusive (Clean Exclusive)	Owner = {P}, Sharers = {}
Shared	Owner = {P}, Sharers = {Any number of P}
Invalid	Owner = {}, Sharers = {}

Table 2: Correspondence between cache state and directory data structure

It might also be necessary for the directory to store the owners and sharers separately in order to find out whether processors own or share a block. However, it is not enough to just store the owner and the sharers of the block, because there are situations where the directory is in the middle of a transaction, and the owner and sharers of the block has not actually received the block, yet. Such additional states can be used to indicate that the directory is busy and cannot fulfill the request at the moment.

Table 2 lists the possible data that the directory can hold compared to the cache state in any cache that has the block. In both the Modified state and the Exclusive state, the directory would store only the node ID of the processor that is the owner. Nevertheless, the directory does not know for certain whether or not the owner is actually clean or dirty until it has sent a request to the cache. It is the job of the protocol to take advantage of having the Exclusive state by not writing the block to memory if the block is clean. If the directory is in a Shared state, then it would have a node ID in the owner slot and any number of node IDs in the sharers list. The reason that there would be a node ID in the owner slot is because any transition into the Shared state must have transitioned from a Modified or Exclusive state, which must already have stored a node ID in the owner slot. If the state of the block is Invalid, there would be no owner or sharers.

One of the easiest ways to keep track of nodes inside directories is to keep a bit vector in each distributed directory about which CPU has which cache block. 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 limit on how many CPUs can be used together with the system since the bit vector would grow too large in a system with too many CPUs. 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 might be unable to keep track of all the CPUs 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.

Keeping track of the owner could be done by either adding a bit to each directory entry, and turning on the extra bit to indicate ownership. There could also be an extra entry dedicated to holding the owner. The advantage of adding a bit to each directory entry that holds the node ID is that it could potentially save bits if there are many nodes in the system and each node ID list is short. However, using an extra entry strictly for holding the owner is more intuitive, and it can also be faster to access since the processor always know which entry is the owner. We can also have a combined owner-sharers list, where one entry implies Exclusive, and multiple entries imply that the directory is in the Shared state. In this last method, the system would have to be able to quickly scan through all the entries it has to determine whether it has no valid entries, one valid entry, or multiple valid entries. Since in either of the protocols that will be tested in this report, the owner entry becomes essentially an additional entry for the sharers list, there is not truly a need for the directory to have different slots for holding the owner versus a list of sharers. Therefore, sometimes in the discussion of protocols in this report, sharers will imply both the sharers list and also the node ID in the owner slot.

Message Type	Message Content	Message Semantics
Read	isExclusive Address	A cache is requesting from the directory to read a block.
Intervention	isExclusive Address	The directory is requesting a block from the exclusive owner.
Read Reply	isExclusive Address BlockData	A reply with data from the home node. The cache will be filled with the block received from the read reply.
Read Response (Origin)	isExclusive Address BlockData	A response with data from the previous exclusive owner with data. The read response moves the data from the previous owner to the new owner or sharer and signifies that the new owner should use the data in this response to fill its cache.
Read Ack	isExclusive Address	An ack from the previous owner or the home node to the new owner without data.
Memory Return	isExclusive Address BlockData	A read response message from the memory to the directory with the block attached.
Speculative Reply (Origin)	Address BlockData	A speculative reply from the directory to the cache. The new owner with fill its cache with data from this message if the other response message it receives is a read ack and not a read response.
Writeback Request	Address BlockData	The cache is in Dirty Exclusive and wants to evict a block.
Eviction Request (BIP)	Address BlockData	The cache is in Clean Exclusive and wants to evict a block.
Writeback Ack (BIP)	Address	Sent from directory to writeback requester to indicate that writeback was received.
Writeback Ack (Origin)	isBusy Address	Sent from directory to writeback requester to indicate that writeback was received.
Eviction Ack (BIP)	Address	The directory sends this message to the requester to signal a successful eviction.
Writeback	Address BlockData	If the cache is asked to give up a block and it is in Dirty Exclusive, it will send this message with the block attached to the directory.
Transfer	isDirty Address	The cache is being asked via an intervention message to give up a block, but it does not need to send the block data to the directory.
Invalidate (BIP)	Address	The cache is being asked by the directory to invalidate its block in the Shared state.
Invalidate (Origin)	NewOwner Address	The cache is being asked by the directory to invalidate its block in the Shared state.
Invalidate Ack	Address	This is sent from the cache to either the new owner or the directory to indicate that the cache received the invalidate.
Nak	isCache Address	A nak is sent whenever the cache or the directory is busy and cannot process the current request.

Table 3: All message types used in BIP and Origin protocol

4 Message Types

Table 3 shows the message types used in the system. Messages sent in the network all need to contain the message type and the address. The message type is necessary for the receiver to know what operations it should take with the message and the address is used to find the requested information. It can also contain data if it is a response to a request. To begin with, every cache coherence system needs a read request message type. In a directory-based cache coherence system, this message would be sent from the cache to the directory. There are two types of read request messages, exclusive and shared. The exclusive read request message is sent when the cache receives a write from the processor, and the shared read request occurs if the cache receives a read from the processor.

Another message that is similar to the read request message is the intervention message, which is used by both BIP and Origin protocol to indicate when a read request is sent from the directory to a cache that intervenes with the cache's normal operation and requests that it provides a block. This occurs when the directory decides that the current owner, depending on whether it owns the block in Dirty Exclusive or Clean Exclusive, should evict its block or send a copy in response to the request. In BIP, the response from the current owner is sent to the directory, whereas the response is sent to the original requester in Origin protocol. When a shared intervention request is received, the current owner transitions to the Shared state, whereas the current owner transitions to the Invalid state if an exclusive intervention request was received. After sending the read request message, the sending cache would wait for a response to the read request.

The five read response types used amongst the two protocols are read response, read reply, read ack, speculative reply, and memory return. A read response indicates a response from another cache, whereas the read reply message indicates a response from the memory or the home node. Since the BIP only supports bilateral interactions between a cache and a directory, and never between two caches, there are no read response messages sent in the BIP protocol. It is also possible for a read ack to be sent, in the case where the block does not need to be attached. Another type of read response message is the memory return message type. This message can only be emitted from the main memory, and it is only used to send a message from the memory to the directory when the directory requests for data. For all of the read response message types mentioned above, the message can be additionally qualified with whether or not it is a response to an exclusive or a shared request. The final read response type is the speculative reply, used only by the directory for sending a block reply in the Origin protocol. The speculative reply is used when the directory does not know in advance whether or not the owner of the block is in Clean Exclusive or Dirty Exclusive, and the directory expects the owner to send its block to the requester, as well, if it was in the Dirty Exclusive state.

The third message type is the writeback request message type. There are two types of writeback requests: writeback request and eviction request. Writeback request is sent when the cache needs to evict a dirty block, but it needs to notify the directory and wait for a response before it can evict it without causing any possible errors in coherency. Eviction request is sent when the cache needs to evict a clean block, and it is only used in BIP because the directory in a Origin protocol does not need to know when a clean exclusive block is evicted.

After sending a writeback request or an eviction request, writeback responses need to be sent to the requesters. The three types of writeback responses are writeback exclusive ack, writeback busy ack and eviction ack. Writeback busy acks are only used in the Origin protocol and eviction acks are only used in BIP. Regular writeback exclusive acks are sent normally by the directory after receiving a writeback request. However, in the Origin protocol, a writeback busy ack would be used if an intervention message arrives after a writeback request was sent. Eviction acks are sent in response to eviction requests.

Besides the writeback request messages, there are also the writeback and transfer messages. These messages are used whenever an intervention message is sent by the directory to the previous owner. The previous owner needs to notify the directory that it has received the intervention, and it will do so by sending either a writeback message or a transfer message. A writeback message has the block attached and indicates that the directory should write the block to memory. A transfer message can be either dirty or shared, depending on whether or not the request was exclusive or shared, but it would have no data in either case.

The next message type is invalidate. This message is sent whenever an exclusive request arrives when the cache block is in the Shared state. In this case, the directory needs to invalidate all of the shared block by sending invalidates to all of them. The caches that receive the invalidate have to send an invalidate ack after invalidating their cache. In the Origin protocol, the invalidates are required to carry a NewOwner tag so that the receiver knows where the invalidate acks should be sent. In BIP, invalidate acks always go to the home node, which is calculated from the address.

The final message type is the nak message. The nak message is further divided into directory nak message and cache nak message, since it is important for the hub to know whether to send the nak message to the directory or the cache. The nak message is sent whenever the directory or the cache is busy, and it cannot process the request at the time. The component that receives a nak message would retry its request[1][2].

5 Cache Coherence Protocol

In this section, I will be going into protocol details. There are several different types of cache coherence protocols, and each of them provide their own advantages and disadvantages. I will first introduce the snoopy-based cache coherence protocol. Next, I will present directory-based cache coherence protocols in general and also show a directory-based cache coherence protocol where all requests and replies pass through the home node and each node can only emit one response in reply to a request. Then, I will be showing a more optimized version of the cache coherence protocol, based on the SGI Origin protocol, where it is possible for each node to emit two messages in response to a request, and not all requests have to pass through the directory before being satisfied.

An interesting aspect about the finite-state machines (FSMs) shown in this section is that they contain many more states than previous sections would indicate. However, the system does not actually store this many states in the system. The reason that there appear to be many more states than there are when the directory and the cache was described in Section 3 is because many of the states in the FSM diagrams are almost mirror-images of each other. For example, in many cases, the transitions for a shared request versus an exclusive request are the same, except shared responses are emitted for the former and exclusive messages are sent for the latter. These transitions and states are shown separately on the state diagram to avoid having too many transitions out of any state in the diagram. In addition, the previous section shows that many messages include an exclusive bit or a dirty bit, which can also reduce the number of states stored as the sender does not need to remember how these bits are set in the messages that it sent.

Some examples of the above mechanisms at work can show that the directory and the cache does not need to actually store all the individual states shown in the diagrams. One instance is that the directory knows when it is in the Unowned state when there are no sharers or owners for an address. Similarly, the directory can tell when it is in the Shared state or Exclusive state by counting the number of nodes that has the address. If there is only one node with ownership, then the directory knows that it is in one of the exclusive states. If there are more than one sharers, then the directory knows that it is in one of the shared states. However, it is necessary to have a busy bit to indicate when the directory is busy processing some request. The cache has comparable properties. The cache has to keep some information to indicate whether or not the cache is clean or dirty, and whether or not it has exclusive ownership of the block. The MESI cache requires at least two bits per cache line to store its four states. Like the directory, it needs a busy/waiting bit to indicate that the cache is processing some request and cannot respond to any requests, but it does not need to store all the states that are shown in the FSMs presented below.

5.1 Snoopy-Based Cache Coherence Protocol

There are many cache coherence protocols that are snoopy-based. A simplified diagram of a snoopy-based cache coherence protocol is shown in Figure 3. The snoopy-based cache coherence protocol relies on the bus to transfer the necessary information. In this protocol, there are only two logical nodes, the requesting node and the data node. However, even though the nodes only need to communicate directly with each other, because they are connected by a bus, all messages are broadcast. These messages are monitored by each processor and its cache; therefore, each processor needs a controller to snoop the bus. Anytime a node encounters an important message that pertains to itself, the controller makes sure to forward the request to the cache so that it can further process it. When the processor requires something, it sends its request on the bus, and this broadcast message is heard by every processor connected to the bus.

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; therefore, manufacturers can easily convert single-core processors to be used as multiprocessors. However,

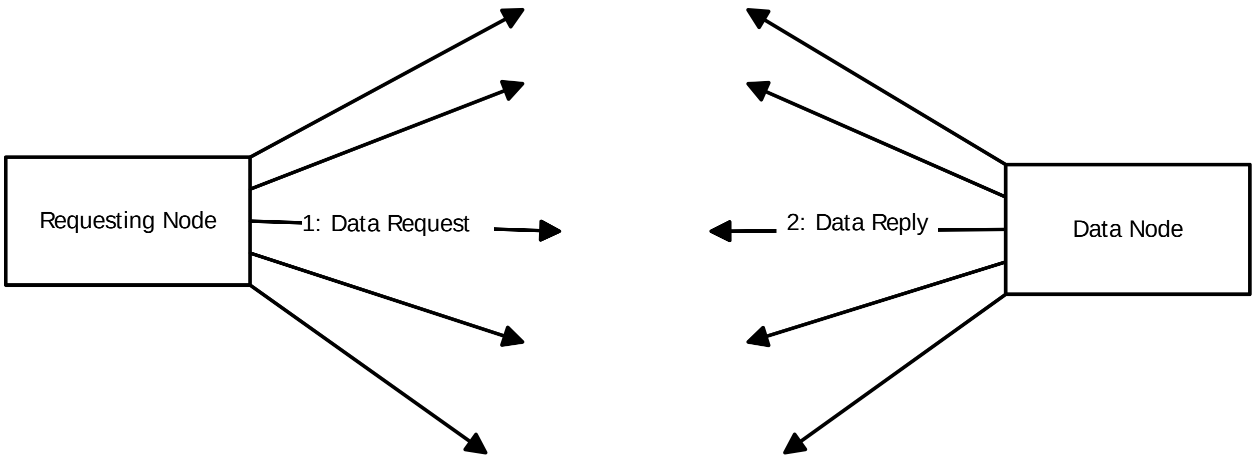


Figure 3: Snoopy-based cache coherence protocol

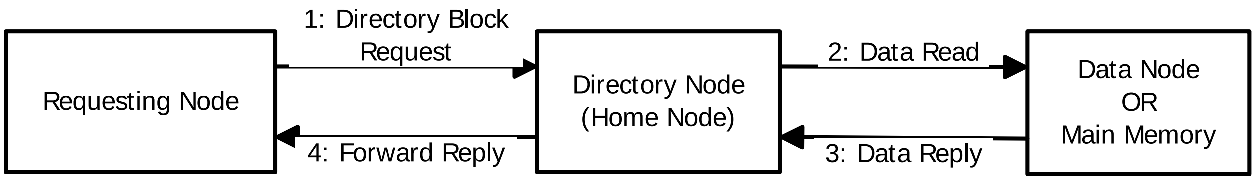


Figure 4: Protocol based on bilateral interactions

snoopy-based cache coherence protocols have more problems with scaling up to meet higher core counts than directory-based cache coherence protocols[2]. The main problem with scaling up snoopy-based protocols is that all the processors are sharing the same medium, the bus. Unless the bandwidth and speed of the bus can be scaled infinitely high, the number of processors cannot be scaled arbitrarily high. For that reason, this report will be focusing mainly on directory-based cache coherence protocols.

5.2 BIP

Directory-based cache coherence protocols such as the BIP 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"[2]. Directory-based protocols are interesting because it can achieve higher speeds performing the same amount of work than snoopy-based protocols. Nevertheless, there are still some characteristics of directory-based cache coherence protocols that can cause problems when designing a system. A snoopy-based protocol has very few problems with consistency because all messages are broadcast on a bus, meaning all processors know whenever changes occur 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 each node knows right away if it needs to change its cache 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 arrive at the directory, allowing the directory to possibly fetch the invalidated data from the owner that just evicted its data if care is not taken when designing the protocol.

A simple request of a directory-based cache coherence protocol based on BIP is shown in Figure 4. The BIP is a traditional directory-based cache coherence protocol where the requester first sends a message to the directory, followed by the directory sending another request to the data node, with the data node returning a response to the directory. Finally, the directory is responsible for forwarding the response back to the requester. In general, two bilateral interactions occur in this protocol to form a complete transaction. In the

Directory State	Semantics
Unowned	No cache owns this block.
Exclusive	One cache owns this block in "Dirty Exclusive" or "Clean Exclusive."
Shared	More than one cache holds the block with read-only access.
Exclusive Waiting for Reply	The directory has received a read request and is waiting for the data to arrive from local memory.
Shared Waiting for Reply	The directory received a shared read request in "Shared" and is waiting on a reply before sending out replies to the requester.
Exclusive Shared Waiting for Reply	The directory received a shared request while it was in "Exclusive," so it is waiting on the current owner to provide the most updated version of the block for further processing.
Waiting to Send Invalidates	The directory received an exclusive read while in "Shared," and is waiting on a reply before sending out invalidates to the rest of the sharers.
Waiting for K Invalidates, J Invalidates Received So Far	After sending out all the invalidates, the directory waits for the same number of invalidate acks to return to make sure that all previous sharers are invalidated before sending out the exclusive reply to the requester.
Waiting for Writeback	The directory is waiting for any type of writeback message, which includes writeback request, eviction request, writeback, and transfer, because an exclusive read was received from another owner while the directory is in "Exclusive."

Table 4: BIP directory states

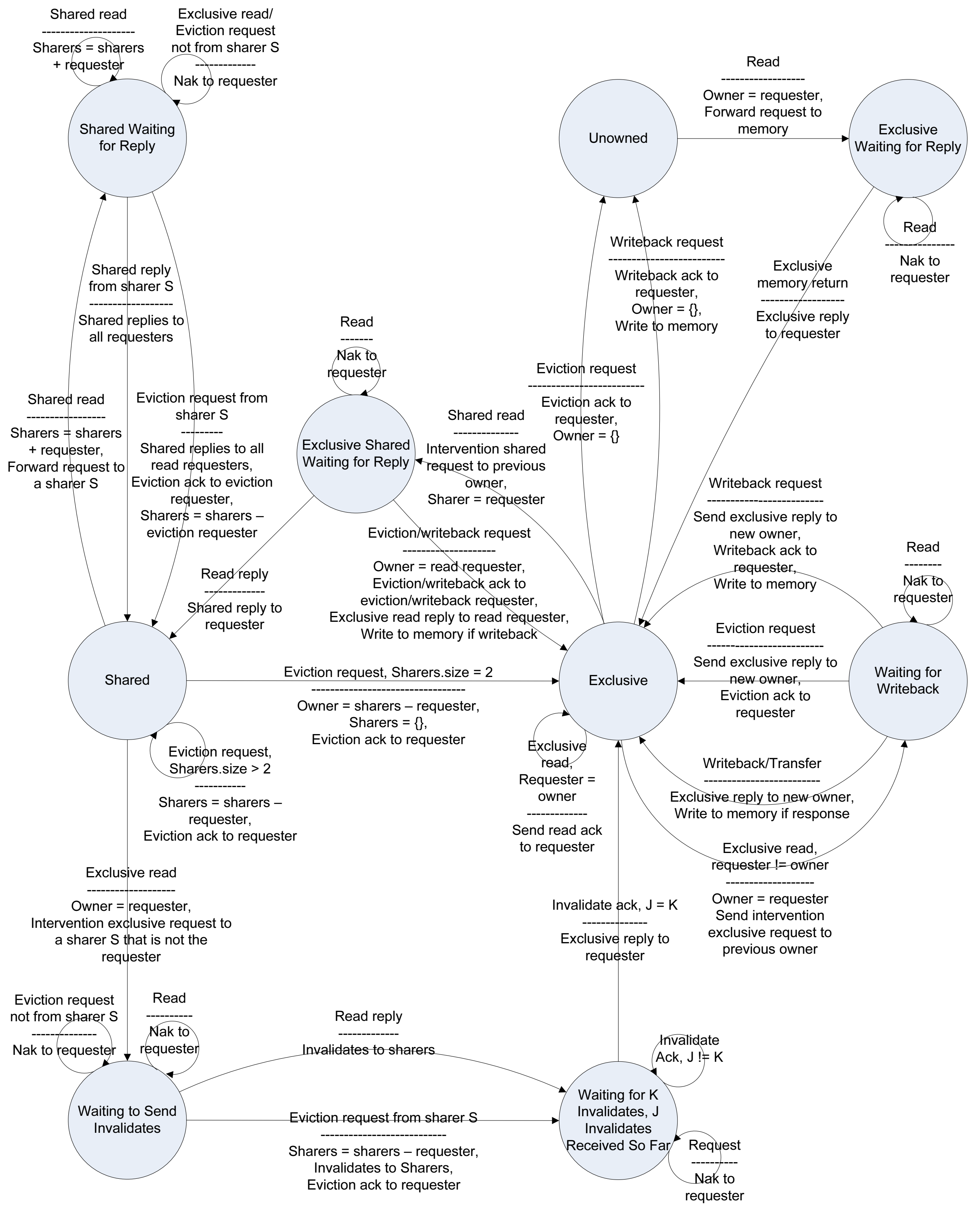
BIP, there are three logical nodes in any request: the requesting node, the directory node, and the data node. These logical nodes can all be the same physical nodes or they can all be different, but it is easier to think of them logically as three separate nodes. Requesting node is the node that sends out the original read or write request. Like in any other directory-based protocol, it is necessary to send the request to the directory node 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. In the BIP, all requests pass through the home node and each node can only emit one response in reply to a request.

This 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 only one corresponding reply, whereas a different protocol may allow more than one response message to be sent back to the original requester for a single request. The requested block need to return to the directory before we forward it back to the requester.

5.2.1 BIP Directory FSM

Figure 5 illustrates the directory FSM for the BIP and Table 4 provides the corresponding states in a table. The directory for any address starts in the "Unowned" state initially. When the directory receives a read request, it would set requester as the owner, forward the request to the memory node, and transition to "Exclusive Waiting for Reply." In that state, the directory has to wait for a response from the memory node before it can do anything else, so any read requests that arrive gets nakked. No writeback requests can arrive in "Unowned" or "Exclusive Waiting for Reply," because no node has the block, yet. When the directory receives a reply from the memory node in "Exclusive Waiting for Reply," an exclusive reply is sent to the requester, and the directory transitions into the "Exclusive" state.

From the "Exclusive" state, there are a myriad of situations that can occur. The first can happen when an exclusive read request arrives at the directory and the requester is the owner. This situation occurs when the cache has the block in its "Shared" state, but it wants exclusive access to it for a write. The directory has to send a read ack (no data) to the requester when this happens. The second can happen when a writeback request is emitted from the owner. In this case, the directory will clear the owner list, send the


Shared read

Sharers = sharers + requester
Exclusive read/ Eviction request not from sharer S

Nak to requester
Shared Waiting for Reply
Read

Owner = requester, Forward request to memory
Unowned
Exclusive Waiting for Reply
Read

Nak to requester
Writeback request

Writeback ack to requester, Owner = {}, Write to memory
Exclusive memory return

Exclusive reply to requester
Shared reply from sharer S

Shared replies to all requesters
Read

Nak to requester
Eviction request

Eviction ack to requester, Owner = {}
Shared read

Sharers = sharers + requester, Forward request to a sharer S
Eviction request from sharer S

Shared replies to all read requesters, Eviction ack to eviction requester, Sharers = sharers – eviction requester
Exclusive Shared Waiting for Reply
Shared read

Intervention shared request to previous owner, Sharer = requester
Writeback request

Send exclusive reply to new owner, Writeback ack to requester, Write to memory
Read

Nak to requester
Eviction/writeback request

Owner = read requester, Eviction/writeback ack to eviction/writeback requester, Exclusive read reply to read requester, Write to memory if writeback
Read reply

Shared reply to requester
Eviction request

Send exclusive reply to new owner, Eviction ack to requester
Shared
Eviction request, Sharers.size = 2

Owner = sharers – requester, Sharers = {}, Eviction ack to requester
Exclusive
Waiting for Writeback
Eviction request, Sharers.size > 2

Sharers = sharers – requester, Eviction ack to requester
Exclusive read, Requester = owner

Send read ack to requester
Writeback/Transfer

Exclusive reply to new owner, Write to memory if response
Exclusive read, requester != owner

Owner = requester Send intervention exclusive request to previous owner
Exclusive read

Owner = requester, Intervention exclusive request to a sharer S that is not the requester
Invalidate ack, J = K

Exclusive reply to requester
Eviction request not from sharer S

Nak to requester
Read

Nak to requester
Read reply

Invalidates to sharers
Invalidate Ack, J != K
Waiting to Send Invalidates
Eviction request from sharer S

Sharers = sharers – requester, Invalidates to Sharers, Eviction ack to requester
Waiting for K Invalidates, J Invalidates Received So Far
Request

Nak to requester

Figure 5: BIP directory FSM

write to memory, send a writeback ack to the requester, and transition to "Unowned." The third is when the directory receives an eviction request in this state. The directory is required to send an eviction ack to the requester, clear the owner list, and transition to "Unowned." It is not necessary to write to memory in this case, because an eviction request indicates that the message came from a cache whose state was "Clean Exclusive."

Another type of transition that occurs from the "Exclusive" state is when the directory receives an exclusive read and the requester is not the owner. In this situation, the directory would set the requester as the owner and send an intervention exclusive request to the previous owner, while transitioning to "Waiting for Writeback," where it waits for a writeback request, an eviction request, a writeback, or a transfer to cause it to transition back to "Exclusive." If a read request arrives while the directory is in "Waiting for Writeback," the directory would send a nak to the requester. If the directory receives a writeback request, it would send an exclusive reply to the new owner, a writeback ack to the requester, send a write to memory, and transition to "Exclusive." On the other hand, if only an eviction request was received, the directory would send an exclusive reply to the new owner, send an eviction ack to the requester, and transition to "Exclusive" without sending a write to memory. In "Waiting for Writeback," the directory could also receive a writeback or a transfer message. In both cases, the directory would send an exclusive reply to the new owner and transition to "Exclusive," but it would only send a write to memory if it received a writeback, indicating that the cache was in the "Dirty Exclusive" state.

It is also possible to transition from the "Exclusive" state to "Exclusive Shared Waiting for Reply" when the directory receives a shared read. This shared read has to come from a requester that is not the owner. The directory has to send an intervention shared request to the previous owner and add the requester as a sharer. In "Exclusive Shared Waiting for Reply," several things can happen. If the directory receives a read, it would send a nak to the requester. If a writeback request arrives from the previous owner, then the directory would set the owner to be the read requester, send a writeback ack to the writeback requester, send an exclusive read reply to the read requester, send a write to memory, and transition back to "Exclusive." Receiving an eviction request is similar to receiving a writeback request, except the directory does not have to send a write to memory. Otherwise, if it receives a read reply from the previous owner, then the directory would forward a shared reply to the requester and transition to "Shared."

In the "Shared" state, if the directory receives an eviction request from a sharer and there are more than two sharers, then the directory would simply remove the requester from the sharers list and send an eviction ack to the requester. If the directory receives a writeback request and the size of the sharers list is 2, then the directory would set the owner to the remaining sharer, clear the sharers list, send an eviction ack to the requester, and transition to "Exclusive."

When a shared read arrives in the "Shared" state, the directory would add the requester into the sharers list, forward the request to a sharer S, which can be any node among the available sharers, and transition to "Shared Waiting for Reply." In this state, the directory adds any new shared requesters into the sharers list and naks any exclusive reads and eviction requests that are not from sharer S. If the directory receives a shared reply from sharer S in this state, it would send shared replies to all requesters. However, if an eviction request from sharer S was received, then the directory would send shared replies to all read requesters, send an eviction ack to the eviction requester, remove the eviction requester from the sharers list, and transition back to "Shared."

While in "Shared" state, it is also possible to receive an exclusive read request. This request causes the requester to be set as the owner, a transition to "Waiting to Send Invalidates," and an intervention exclusive request to be forwarded to a sharer S, which can be any node that is not the requester. In the "Waiting to Send Invalidates" state, if the directory receives an eviction request not from sharer S, it would send a nak to the requester. Likewise, if the directory receives a read request in this state, it would send a nak to the requester. If a read reply arrives from sharer S, the directory would send invalidates to all sharers and transition to "Waiting for K Invalidates, J Invalidates Received So Far," where K is the number of invalidates sent, and J is the number of invalidate acks received so far. However, if an eviction request from sharer S arrives, then the directory would remove the requester from the sharers list, send invalidates to the remaining sharers, send an eviction ack to the requester, then transition to "Waiting for K Invalidates, J Invalidates Received So Far."

The directory waits for invalidate acks in the "Waiting for K Invalidates, J Invalidates Received So Far" state. If a read request or an eviction request was received, the directory has to send a nak to the requester.

Cache State	Semantics
Invalid	The cache does not own the block.
Clean Exclusive	The cache owns the block with exclusive access, but it has not modified the block.
Dirty Exclusive	The cache owns the block with exclusive access, and it has modified the block.
Shared	The cache holds the block without exclusive access. It cannot write to it.
Waiting for Shared Read Reply	The cache has just sent a shared read request and is waiting for a response from the directory.
Waiting for Exclusive Read Reply	The cache has just sent an exclusive read request and is waiting for a response from the directory.
Clean Exclusive Waiting for Eviction Ack	The cache has evicted a clean block and is waiting for the directory to approve the eviction.
Dirty Exclusive Waiting for Writeback Ack	The cache has evicted a modified block and is waiting for the directory to approve the writeback.
Shared Waiting for Eviction Ack	The cache has evicted a block in the "Shared" state and is waiting for approval from the directory.
Waiting for Nak	An invalidate was received when the cache was waiting for an eviction ack, so the cache has to wait for a nak before sending an invalidate ack.

Table 5: BIP cache states

If an invalidate ack was received, the directory checks if J (the number of invalidate acks received so far) is equal to K (the number of invalidates sent). If they are not the same, then the directory continues to wait in this state. However, if J = K, then the directory would send an exclusive reply to the requester and transition to "Exclusive."

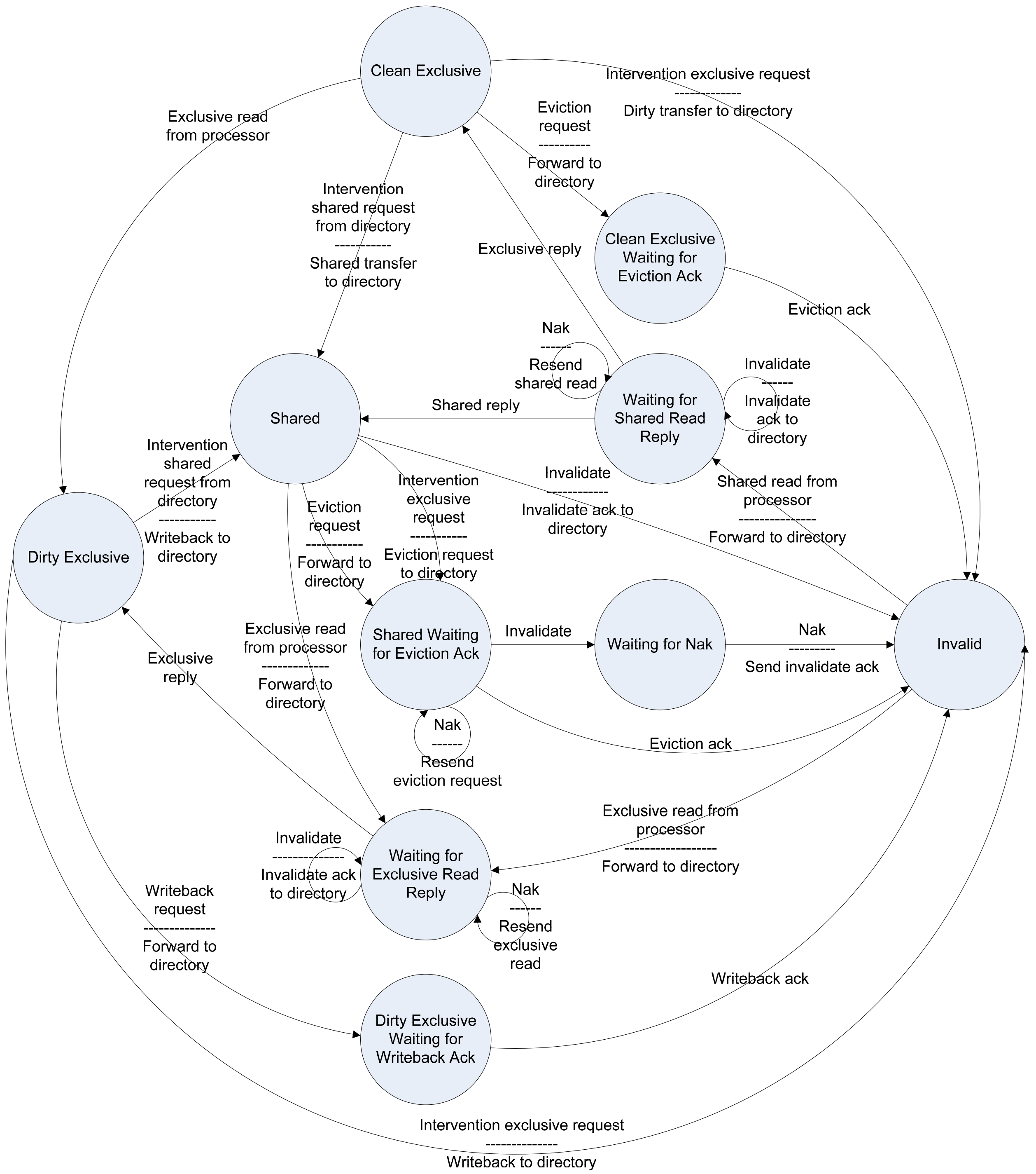
5.2.2 BIP Cache FSM

Figure 6 illustrates the cache FSM for the BIP and Table 5 shows all the corresponding cache states. All caches start initially in the "Invalid" state. From this state, the cache can receive a request from the processor for either a shared read or an exclusive read. If the cache receives a shared read, then it would transition to "Waiting for Shared Read Reply" after forwarding the request to the directory. However, if the cache receives an exclusive read, it would transition to "Waiting for Exclusive Reply" after forwarding the request to the directory.

In "Waiting for Shared Read Reply," the cache waits for either type of read replies and transitions depending on which type of read reply it receives. If it receives a nak, it has to resend the shared read request. If the cache receives an invalidate, it has to send an invalidate ack to the directory. The cache would fill the cache with the block and transition to "Shared" if it receives a shared reply. However, if the cache receives an exclusive reply, it would transition to "Clean Exclusive," instead.

Once in "Clean Exclusive," the cache can receive an intervention exclusive request from the directory, when it has to reply with a dirty transfer and transition to "Invalid." The cache could also receive an intervention shared request from the directory, indicating that the cache should transition into the "Shared" state and send a shared transfer to the directory. In addition, the cache could decide to evict the block by sending an eviction request to the directory and transitioning to "Clean Exclusive Waiting for Eviction Ack," where it waits for an eviction ack before transitioning to "Invalid." Otherwise, the cache can also transition into the "Dirty Exclusive" state if it receives an exclusive read request from the processor.

After arriving in the "Dirty Exclusive" state, several things can happen. The cache can receive an intervention exclusive request from the directory, when it would have to transition to "Invalid" and send a writeback to the directory. If the cache decides to evict the block while in "Dirty Exclusive," it would send



Clean Exclusive
Intervention exclusive request ------------- Dirty transfer to directory
Exclusive read from processor
Eviction request ---------- Forward to directory
Intervention shared request from directory ----------- Shared transfer to directory
Clean Exclusive Waiting for Eviction Ack
Exclusive reply
Eviction ack
Nak ------ Resend shared read
Invalidate ------ Invalidate ack to directory
Shared
Shared reply
Waiting for Shared Read Reply
Intervention shared request from directory ----------- Writeback to directory
Dirty Exclusive
Eviction request ----------- Forward to directory
Intervention exclusive request ----------- Eviction request to directory
Invalidate ----------- Invalidate ack to directory
Shared read from processor -------------- Forward to directory
Invalid
Exclusive reply
Exclusive read from processor ------------- Forward to directory
Shared Waiting for Eviction Ack
Invalidate
Waiting for Nak
Nak --------- Send invalidate ack
Nak ------ Resend eviction request
Eviction ack
Exclusive read from processor ----------------- Forward to directory
Invalidate -------------- Invalidate ack to directory
Waiting for Exclusive Read Reply
Nak ------ Resend exclusive read
Writeback request -------------- Forward to directory
Writeback ack
Dirty Exclusive Waiting for Writeback Ack
Intervention exclusive request -------------- Writeback to directory

Figure 6: BIP cache FSM

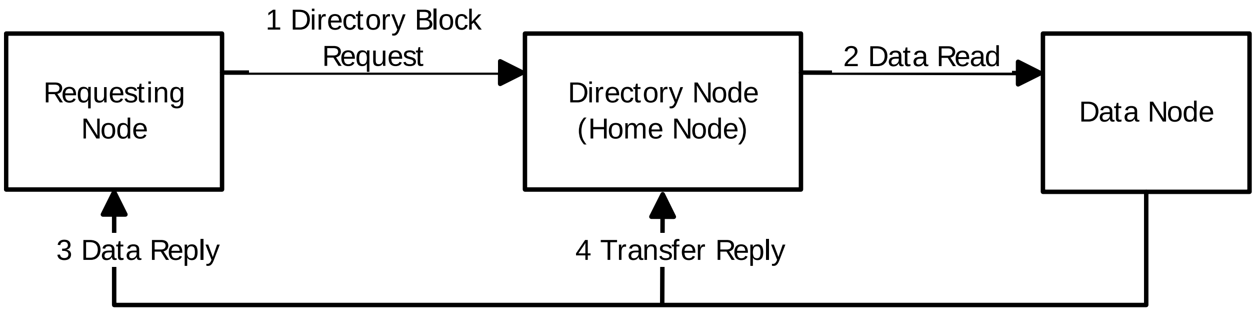


Figure 7: Origin protocol

a writeback request to the directory and transition into "Dirty Exclusive Waiting for Writeback Ack," where it waits for a writeback ack from the directory before transitioning to "Invalid." In "Dirty Exclusive," the cache could also receive an intervention shared request from the directory, indicating that the cache should transition into the "Shared" state and send a writeback to the directory.

Once the cache transitions into the "Shared" state, several transitions could happen. The directory could receive an invalidate message from the directory, indicating that the cache should transition to "Invalid" and send an invalidate ack to the directory. If this cache received an invalidate, then this cache could not have been the sharer S that the directory chose to read from, since the directory is required to obtain the first read reply from sharer S before sending out its invalidates. In addition, if the cache has to evict a block in the "Shared" state for any reason, it would send an eviction request to the directory and transition to "Shared Waiting for Eviction Ack." If the cache receives an intervention exclusive message, it would also send an eviction request to the directory and transition to "Shared Waiting for Eviction Ack."

After transitioning to "Shared Waiting for Eviction Ack," the cache can wait for several messages. If the cache receives a nak, it would resend the eviction request. If the cache receives an invalidate message from the directory, then it means that the eviction request has not arrived at the directory before the invalidate message was sent, so it has to transition to "Waiting for Nak," since the directory would send a nak in response to the cache's eviction request. Once the cache receives a nak in "Waiting for Nak," it would send an invalidate ack and transition to "Invalid." It is also possible for the cache to receive an eviction ack while in "Shared Waiting for Eviction Ack," where the cache can transition to "Invalid" without doing anything else.

In "Shared," it is also possible for the cache to receive an exclusive read request from the processor, where it would forward the request to the directory and transition to "Waiting for Exclusive Read Reply." In this state, the cache can receive a nak from the directory, telling the cache that the directory is busy and that it should resend its request. The cache can also receive an invalidate message, which means it has to send an invalidate ack to the directory. Only the exclusive reply message from the directory will transition the cache out of this state and into the "Dirty Exclusive" state.

5.3 Origin Protocol

In this section, I will be showing a more optimized version of the cache coherence protocol. The SGI Origin 2000 implemented this protocol, which is based on an altered version of the protocol used in the Stanford DASH multiprocessor[1]. In this protocol, it is possible for each node to emit two messages in response to a request, and not all requests have to pass through the directory before being satisfied. This allows the Origin protocol to send less messages into the network than BIP. This decreases the overall latency in the system because it eliminates the time going from the data node to the directory node to complete requests. The protocol assumes that all read requests to the directory can be satisfied, and if it cannot, it is up to the owner of the data instead of the directory to send an additional invalidate back to the directory as well as the data response to the original requesting node. We save time by allowing the data packet retrieved to go directly to the requester without the additional latency of going to the directory first.

Figure 7 shows a simple request that could occur in the Origin protocol. In the Origin protocol, there are also three logical nodes, like in BIP, but occasionally, the data node needs to send two messages at once, one to the directory node, and one to the requesting node. This type of protocol introduce many opportunities for the messages to arrive out-of-order. A requesting message goes to the directory, which can be different than the node that ultimately supplies the response back to the requester. There are situations where the directory completes its transition into a non-busy state before knowing for certain that the new owner has received its request, making it possible for the directory to send another request to the new owner when the new owner has not even received its response from the previous transaction. In addition, the system sometimes need to send out two messages simultaneously, as opposed to the BIP, where all operations can only cause one outgoing message to be sent on every incoming message. The protocol deals with this situation by having the directory change to a busy state whenever the directory state and the request cannot be satisfied immediately. For example, this situation can happen when the directory state is Exclusive with another owner, and a read request arrives. Any further requests that the directory receives, regardless of their origin or type, will be denied via a nak by the directory. This ensures that the directory stays in a consistent state and is not modified based on an invalid directory state.

This protocol is also different from BIP in the way it handles invalidations. Invalidates have to be sent when transitioning out of the shared state into the exclusive state or the invalid state. When this occurs in BIP, the invalidate acks return to the directory, and the directory would only send out a response to the read request when it receives all the invalidate acks. If the same thing occurs in the Origin protocol, the invalidate acks return to the new owner. Therefore, the cache is the component that needs to remember the number of invalidate acks received in the Origin protocol. Because of these changes, the finite-state machines for this protocol is substantially different than the BIP, both in the directory and in the cache.

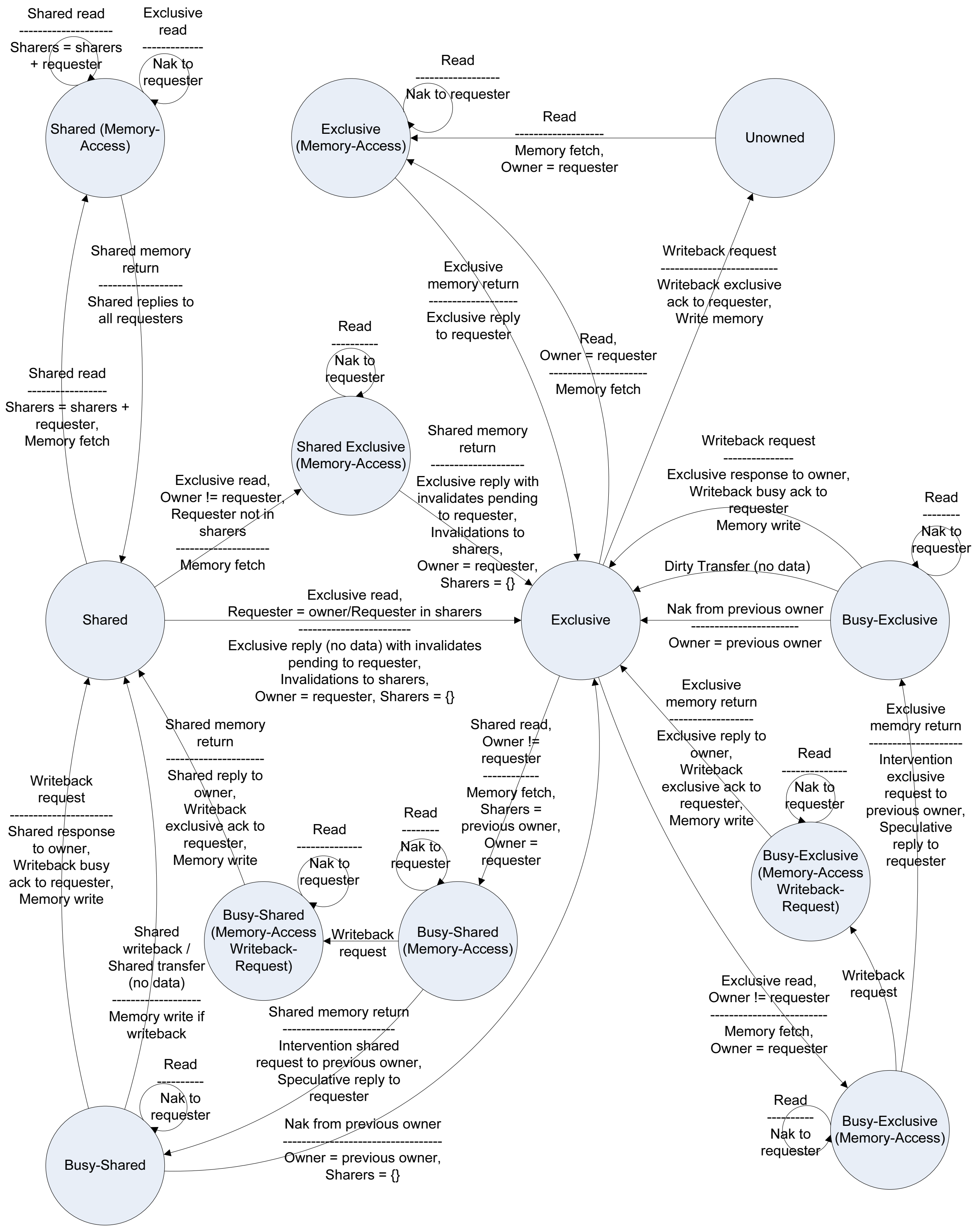
5.3.1 Origin Protocol Directory FSM

Figure 8 illustrates the directory FSM for the Origin directory-based cache coherence protocol and Table 6 shows all the states in the directory. Any block will start in the "Unowned" state. From here, if the directory receives either type of read request, the directory would transition to "Exclusive (Memory-Access)," make requester owner, and perform memory fetch. If the directory receives another read request in "Exclusive (Memory-Access)," the directory would send a nak to the requester. When the request from memory returns in the "Exclusive (Memory-Access)" state, the directory would send an exclusive reply to the requester and transition to the "Exclusive" state. It should be impossible for a writeback request to arrive during "Unowned" or "Exclusive (Memory-Access)" states because no node should have the block in these two states.

From the "Shared" state, there are several states the directory can transition into. The first is "Shared (Memory-Access)", which occurs if a shared read request comes in while in the "Shared" state. During the transition, the directory would perform a memory fetch and add the requester to sharers. In "Shared (Memory-Access)", the directory would queue up any more shared read requests that might come in while sending a nak to any exclusive reads. When the memory returns, the directory will send shared replies to all requesters. There could also be a transition to the "Shared Exclusive (Memory-Access)" state or the "Exclusive" state from the "Shared" state when it receives an exclusive read request, depending on whether or not the requester can be found in the owner or sharers of the directory block. If the requester is the owner or in sharers, then the directory has received an upgrade request, which means that the requester already has the data. Otherwise, the directory has to fetch the block from memory before it can send out invalidates to sharers and previous owner, send out exclusive reply with invalidates pending to the requester, set the requester as the owner, and clear sharers. It is impossible to receive a writeback request in the "Shared" state or any of the states that transitions from the "Shared" state because no node has exclusive access to the block. If more read requests arrive while in the "Shared Exclusive (Memory-Access)" state, then the directory has to send a nak to that request.

In the "Exclusive" state, the home memory could transition to the "Exclusive (Memory-Access)" state if the requester is the owner, while also performing a fetch from memory. It could also transition to "Unowned" if it receives a writeback request. However, from "Exclusive," it could also transition to the two busy memory-access states.

In all of the busy states, naks are sent in response to any requests, since the directory cannot process any additional requests when it is busy. The directory will transition to the busy memory-access states when a read request arrives during the "Exclusive" state and the owner is not the requester. It is necessary, both when transitioning to "Busy-Shared (Memory-Access)" and when transitioning to "Busy-Exclusive (Memory-


Shared read

Sharers = sharers + requester
Exclusive read

Nak to requester
Shared (Memory-Access)
Read

Nak to requester
Exclusive (Memory-Access)
Read

Memory fetch, Owner = requester
Unowned
Shared memory return

Shared replies to all requesters
Exclusive memory return

Exclusive reply to requester
Read, Owner = requester

Memory fetch
Writeback request

Writeback exclusive ack to requester, Write memory
Read

Nak to requester
Shared read

Sharers = sharers + requester, Memory fetch
Shared Exclusive (Memory-Access)
Shared memory return

Exclusive reply with invalidates pending to requester, Invalidations to sharers, Owner = requester, Sharers = {}
Writeback request

Exclusive response to owner, Writeback busy ack to requester
Memory write
Read

Nak to requester
Exclusive read, Owner != requester, Requester not in sharers

Memory fetch
Dirty Transfer (no data)
Shared
Exclusive read, Requester = owner/Requester in sharers

Exclusive reply (no data) with invalidates pending to requester, Invalidations to sharers, Owner = requester, Sharers = {}
Exclusive
Nak from previous owner

Owner = previous owner
Busy-Exclusive
Exclusive memory return

Exclusive reply to owner, Writeback exclusive ack to requester, Memory write
Exclusive memory return

Intervention exclusive request to previous owner, Speculative reply to requester
Shared memory return

Shared reply to owner, Writeback exclusive ack to requester, Memory write
Shared read, Owner != requester

Memory fetch, Sharers = previous owner, Owner = requester
Read

Nak to requester
Writeback request

Shared response to owner, Writeback busy ack to requester, Memory write
Read

Nak to requester
Read

Nak to requester
Busy-Exclusive (Memory-Access Writeback-Request)
Busy-Shared (Memory-Access Writeback-Request)
Writeback request
Busy-Shared (Memory-Access)
Shared writeback / Shared transfer (no data)

Memory write if writeback
Shared memory return

Intervention shared request to previous owner, Speculative reply to requester
Exclusive read, Owner != requester

Memory fetch, Owner = requester
Writeback request
Read

Nak to requester
Nak from previous owner

Owner = previous owner, Sharers = {}
Busy-Shared
Read

Nak to requester
Busy-Exclusive (Memory-Access)

Figure 8: Origin protocol directory FSM

Directory State	Semantics
Unowned	No cache owns this block.
Exclusive	One cache owns the block in “Clean Exclusive” or “Dirty Exclusive.”
Shared	More than one cache holds the block with read-only access.
Exclusive (Memory-Access)	The directory is waiting for a message from local memory in response to a read before transitioning to “Exclusive.”
Shared (Memory-Access)	The directory is busy processing a shared read request received when it was in the “Shared” state. It is waiting on a memory return before transitioning to “Shared.”
Shared Exclusive (Memory-Access)	The directory is busy processing an exclusive read request received when it was in the “Shared” state. It is waiting on a memory return before transitioning to “Exclusive.”
Busy-Exclusive (Memory-Access)	The directory is busy processing an exclusive read request. It is in the memory fetching state and waiting for a writeback request or an exclusive memory return.
Busy-Shared (Memory-Access)	The directory is busy processing an shared read request. It is in the memory fetching state and waiting for a writeback request or a shared memory return.
Busy-Exclusive (Memory-Access Writeback-Request)	The directory is busy processing an exclusive read request. It is in the memory fetching state, but it has already received a writeback request, so it will not send an intervention message when the exclusive memory return arrives.
Busy-Shared (Memory-Access Writeback-Request)	The directory is busy processing a shared read request. It is in the memory fetching state, but it has already received a writeback request, so it will not send an intervention message when the shared memory return arrives.
Busy-Exclusive	The directory is busy processing an exclusive read request. It has sent an intervention exclusive request and a speculative reply, so it is only waiting on a writeback request, a dirty transfer, or a nak.
Busy-Shared	The directory is busy processing a shared read request. It has sent an intervention shared request and a speculative reply, so it is only waiting on a writeback request, a shared writeback, a shared transfer, or a nak.

Table 6: Origin protocol directory states

Access)," to send a memory fetch and set the owner as the requester; however, it is only necessary to set the sharers as the previous owner when transitioning to "Busy-Shared (Memory-Access)" since "Busy-Exclusive (Memory-Access) transitions eventually to the "Exclusive" state. In the "Busy-Shared (Memory-Access)" state, we usually would wait for the memory to return before performing any other operations. If the directory receives a memory return first, then it would send an intervention request to the previous owner, send a speculative reply to the requester, and transition to the "Busy-Shared" state. Should a writeback request occur before the memory return, however, the directory would transition to the "Busy-Shared (Memory-Access Writeback-Request)" state, where we would wait for the memory return and not send any messages that we would send when we transition to the "Busy-Shared" state. In the "Busy-Shared (Memory-Access Writeback-Request)" state, we would wait for the memory return message, then send a shared reply to the owner and a writeback exclusive ack to the requester. Another writeback request could not arrive while in the "Busy-Shared (Memory-Access Writeback-Request)" state because the directory just received a writeback request from the only node that had the block. The "Busy-Exclusive (Memory-Access)" state transitions are similar, except we send exclusive messages instead of shared messages and we transition to exclusive states.

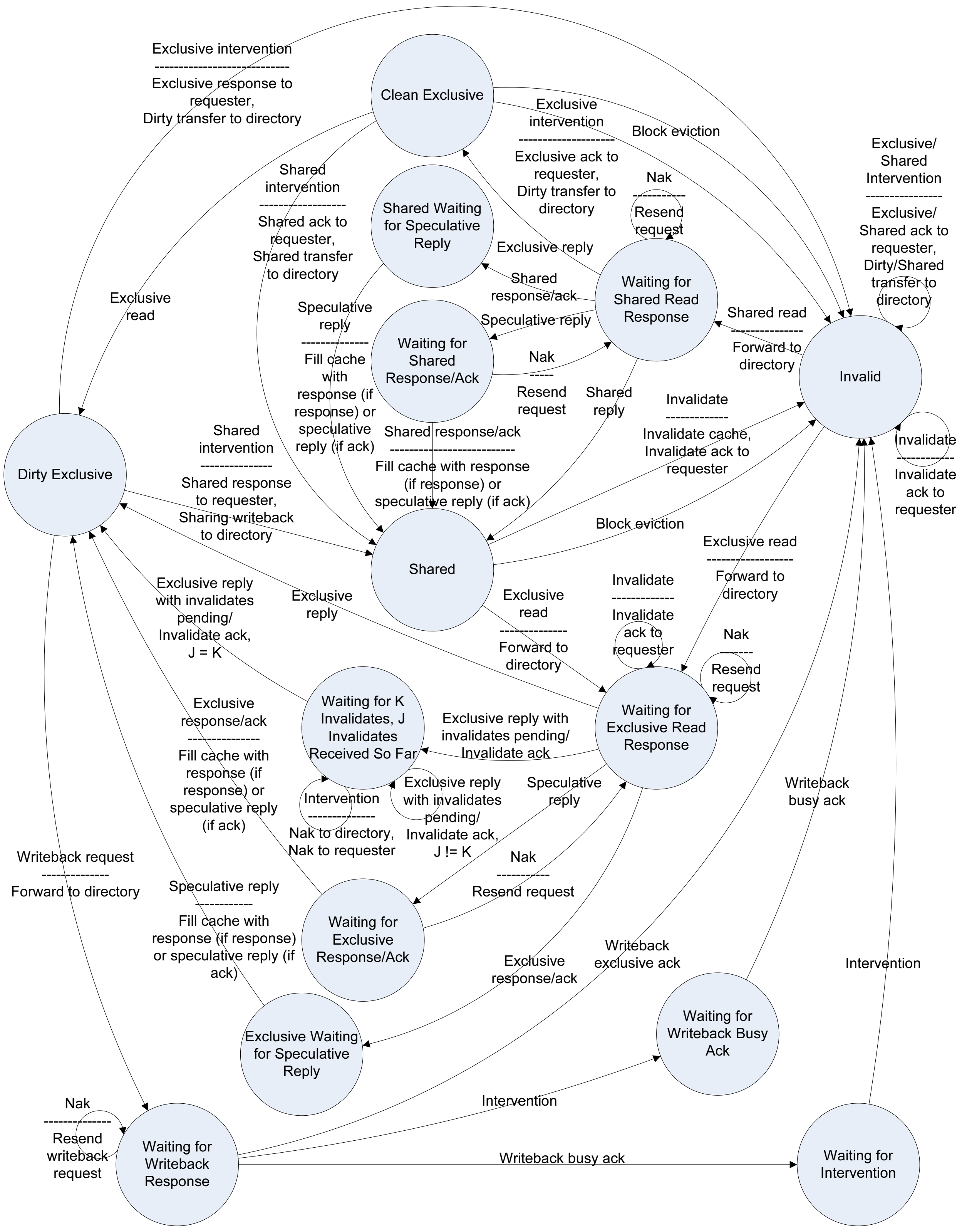
In the "Busy-Shared" state, the directory is waiting for a message from the previous owner of the block before transitioning to the "Shared" state. This could come in the form of a writeback request (if the owner evicted the block before the intervention arrives), a shared writeback, or a shared transfer (no data). If a writeback request or a shared writeback arrives, the directory also has to write the block to memory. Additionally, a writeback request means that the directory would forward that data to the new owner and return a writeback busy ack to the requester. It is possible for the intervention to be unsuccessful, however, which is indicated by a nak from the previous owner. When this happens, the directory would return to "Exclusive" while setting the owner to the previous owner and clear the sharers. The "Busy-Exclusive" state is similar, except exclusive messages are sent instead of shared messages, the directory transitions back to "Exclusive" upon successful completion, and upon receiving a nak in "Busy-Exclusive," it is not necessary to clear the sharers.

5.3.2 Origin Protocol Cache FSM

Figure 9 illustrates the cache FSM for the Origin directory-based cache coherence protocol and Table 7 provides the corresponding states in a table. This diagram shows the state transitions for the cache. Any cache line for an address starts in the "Invalid" state. If the cache receives a shared read from the processor, it transitions to "Waiting for Shared Read Response" and forwards the request to the home memory; if it receives an exclusive read instead, it transitions to "Waiting for Exclusive Read Response." In addition, if the cache receives an intervention, it sends an ack to the requester and a transfer to the directory. The type of the ack and the transfer will be dependent upon whether or not a shared intervention is received or an exclusive intervention is received. If the cache receives an invalidate in the "Invalid" state, then it means that the cache has evicted the block because of a capacity miss, which means that there is no need to invalidate the block again because the cache does not have the block. However, it is still necessary to send an invalidate ack to the requester in accordance with the protocol.

Once the cache transitions to "Waiting for Shared Read Response," it is possible for the cache to receive a nak from the directory, indicating that the directory is currently busy and cannot handle the request, and the cache will have to resend its request. If it receives a shared reply, the cache would transition to the "Shared" state, while receiving an exclusive reply would transition the cache into "Clean Exclusive." If the cache receives an intervention message while in the "Clean Exclusive" state, it has to send an ack to the requester, and send a transfer to the directory, before transitioning to "Shared" or "Invalid," depending on whether or not it was a shared intervention or an exclusive intervention. It is also possible, while in "Clean Exclusive," to receive an exclusive read request, in which case the cache would transition to "Dirty Exclusive." Finally, if the cache needs to evict a block while in "Clean Exclusive," it would transition to "Invalid" without sending any messages.

However, if a cache receives a speculative reply or a shared response/ack in "Waiting for Shared Read Response," then it would transition to "Waiting for Shared Response/Ack" or "Shared Waiting for Speculative Reply," respectively. Receiving either of these messages mean that the directory block state was Exclusive and that the requester needs to wait for both messages before it can fill its cache and transition to the "Shared" state. However, it is possible to get a nak from the previous owner while in the "Waiting for Shared



Clean Exclusive
Shared Waiting for Speculative Reply
Waiting for Shared Response/Ack
Waiting for Shared Read Response
Invalid
Dirty Exclusive
Shared
Waiting for K Invalidates, J Invalidates Received So Far
Waiting for Exclusive Read Response
Waiting for Exclusive Response/Ack
Exclusive Waiting for Speculative Reply
Waiting for Writeback Busy Ack
Waiting for Writeback Response
Waiting for Intervention
Exclusive intervention / Exclusive response to requester, Dirty transfer to directory
Exclusive intervention / Exclusive ack to requester, Dirty transfer to directory
Block eviction
Exclusive/Shared Intervention / Exclusive/Shared ack to requester, Dirty/Shared transfer to directory
Nak / Resend request
Exclusive reply
Shared response/ack
Speculative reply
Shared read / Forward to directory
Shared intervention / Shared ack to requester, Shared transfer to directory
Exclusive read
Speculative reply / Fill cache with response (if response) or speculative reply (if ack)
Nak / Resend request
Shared reply
Invalidate / Invalidate cache, Invalidate ack to requester
Invalidate / Invalidate ack to requester
Shared response/ack / Fill cache with response (if response) or speculative reply (if ack)
Shared intervention / Shared response to requester, Sharing writeback to directory
Block eviction
Exclusive read / Forward to directory
Exclusive reply with invalidates pending/ Invalidate ack, J = K
Exclusive reply
Exclusive read / Forward to directory
Invalidate / Invalidate ack to requester
Nak / Resend request
Exclusive response/ack / Fill cache with response (if response) or speculative reply (if ack)
Exclusive reply with invalidates pending/ Invalidate ack
Intervention / Nak to directory, Nak to requester
Exclusive reply with invalidates pending/ Invalidate ack, J != K
Speculative reply
Writeback busy ack
Writeback request / Forward to directory
Speculative reply / Fill cache with response (if response) or speculative reply (if ack)
Nak / Resend request
Exclusive response/ack
Writeback exclusive ack
Intervention
Nak / Resend writeback request
Intervention
Writeback busy ack

Figure 9: Origin protocol cache FSM

Cache State	Semantics
Invalid	The cache does not own the block.
Clean Exclusive	The cache owns the block with exclusive access, but it has not modified the block.
Dirty Exclusive	The cache owns the block with exclusive access, and it has modified the block.
Shared	The cache holds the block without exclusive access. It cannot write to it.
Waiting for Shared Read Response	The cache is waiting on a response after sending a shared read request to the directory.
Waiting for Exclusive Read Response	The cache is waiting on a response after sending an exclusive read request to the directory.
Shared Waiting for Speculative Reply	The cache has received a shared response or ack, and is waiting on a speculative reply to transition into "Shared."
Exclusive Waiting for Speculative Reply	The cache has received an exclusive response or ack, and is waiting on a speculative reply to transition into "Dirty Exclusive."
Waiting for Shared Response/Ack	The cache has received a speculative reply, and is waiting on a shared response or ack to transition into "Shared."
Waiting for Exclusive Response/Ack	The cache has received a speculative reply, and is waiting on an exclusive response or ack to transition into "Dirty Exclusive."
Waiting for K Invalidates, J Invalidates Received So Far	There are some sharers that need to be invalidated before the block can be used. The cache needs to wait for all invalidate acks to be received before filling with the exclusive reply.
Waiting for Writeback Response	The cache decides to evict a modified block, and is waiting for a response before completing the writeback.
Waiting for Writeback Busy Ack	The cache has received an intervention request, and has to wait for a writeback busy ack to complete the writeback.
Waiting for Intervention	The cache has received a writeback busy ack, and has to wait for an intervention request to complete the writeback.

Table 7: Origin protocol cache states

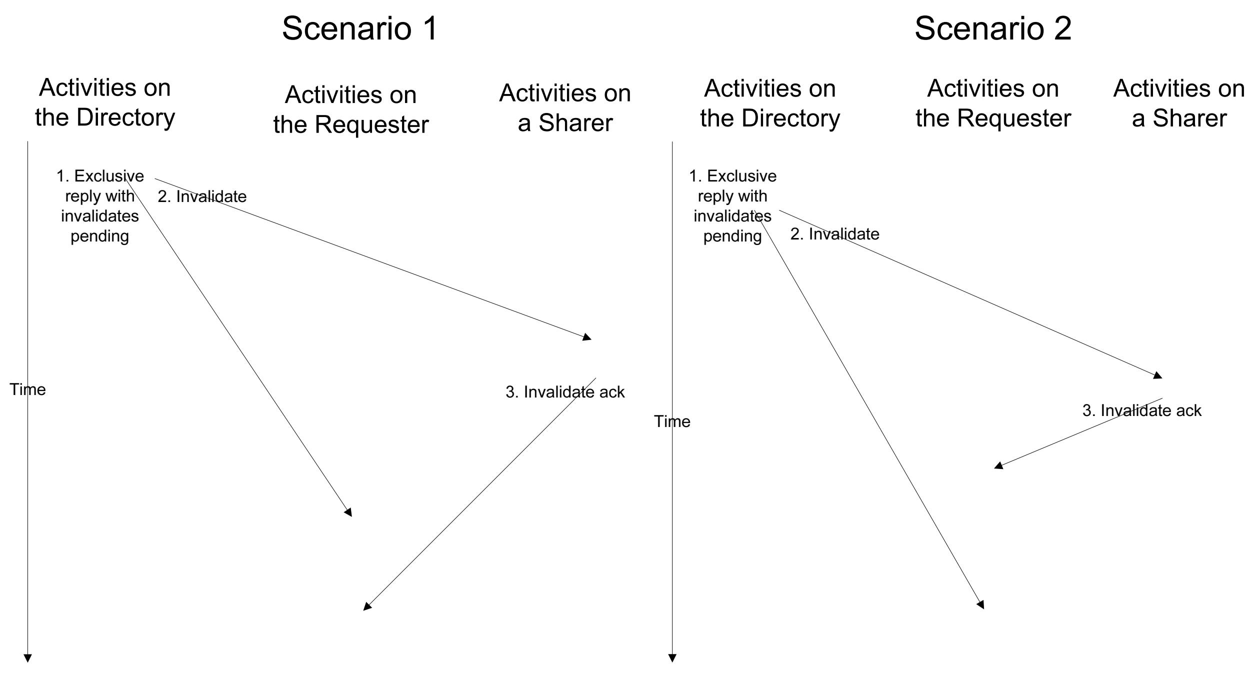

Scenario 1
Activities on the Directory
Activities on the Requester
Activities on a Sharer
1. Exclusive reply with invalidates pending
2. Invalidate
3. Invalidate ack
Time
Scenario 2
Activities on the Directory
Activities on the Requester
Activities on a Sharer
1. Exclusive reply with invalidates pending
2. Invalidate
3. Invalidate ack
Time

Figure 10: Scenarios when transitioning into "Waiting for K Invalidates, J Invalidates Received So Far"

Response/Ack" state, in which case, the cache has to start all over and resend the request while transitioning back to "Waiting for Shared Read Response."

In the "Shared" state, if the cache receives an invalidate from the directory, the cache would have to invalidate the block, send an invalidate ack to the requester, and transition to the "Invalid" state. If the cache needs to evict the block, it would also have to transition to "Invalid," but without sending any messages. It is also possible to receive an exclusive read request from the processor in this state, when the cache would have to forward the request to the directory and transition to "Waiting for Exclusive Read Response." Since the cache might have received the exclusive read request before receiving an invalidate from the directory, it is necessary to send an invalidate ack to the requester when receiving an invalidate in the "Waiting for Exclusive Read Response" state. If the cache receives a nak in "Waiting for Exclusive Read Response," the directory could not fulfill the request and the cache must resend its request. Typically, the cache would receive an exclusive reply in this state and transition to "Dirty Exclusive."

However, if there were any sharers when the cache requested exclusive read, the cache would receive an exclusive reply with invalidates pending or an invalidate ack and transition to "Waiting for K Invalidates, J Invalidates Received So Far." The reason that an exclusive reply with invalidates pending or an invalidate ack could be received from the "Waiting for Exclusive Read Response" state is illustrated in Figure 10. The exclusive reply with invalidates pending and invalidations are sent at the same time to the requester and the sharers, respectively, in both scenarios. In the expected case, scenario 1, the exclusive reply with invalidates pending would arrive at the requester before any invalidate acks. However, if any sharers are on the same node as the directory, it can receive its invalidate before the exclusive reply with invalidates pending arrives at the requester. The sharer would then send an invalidate ack to the requester. Since the invalidate ack has to traverse the network, there is no guarantee that the exclusive reply with invalidates pending would arrive first. Therefore, the requester has to prepare for both scenarios by transitioning into the "Waiting for K Invalidates, J Invalidates Received So Far" state on the arrival of either types of messages. In this state, the cache waits for the arrival of invalidate acks and/or exclusive reply with invalidates pending and transitions to "Dirty Exclusive" only when all invalidate acks and the exclusive reply with invalidates pending have been received. It is also possible to receive an intervention while in "Waiting for K Invalidates, J Invalidates Received So Far," when the cache would have to respond with a nak to the directory and a nak to the requester.

In the "Waiting for Exclusive Read Response" state, it is also possible that the cache has to wait for two

Latency Type	Number of Cycles
L1 Cache Hit	1
L1 Cache Miss	2
L2 Cache Hit	10
L2 Cache Miss	13
Memory Read	57
Memory Write	66
Network Minimum	100
Network Random Range	10
Network Multiplier	0.1

Table 8: Parameters used for simulator configuration

$$T_L = T_m + rand\,(0, T_R) + (T_X \times N)$$

Figure 11: Network latency equation for a single message

messages before it can transition to "Dirty Exclusive." When the cache receives an exclusive response/ack or a speculative reply, it would transition to "Exclusive Waiting for Speculative Reply" or "Waiting for Exclusive Response/Ack," respectively, while it waits for the other message of this two-message exchange to arrive. When the other message arrives, the cache would fill the cache with the exclusive response (if exclusive response) or speculative reply (if exclusive ack) and transition to "Dirty Exclusive." However, it is possible that the cache receives a nak from the previous owner in the "Waiting for Exclusive Response/Ack" state, when the cache would have to start over by resending its request to the directory and transitioning to "Waiting for Exclusive Read Response."

Once in the "Dirty Exclusive" state, the cache can receive three types of messages. If the cache receives a shared intervention, it would send a shared response to the requester and send a shared writeback to the directory while transitioning to "Shared." Similarly, if the message was an exclusive intervention, it would send an exclusive response to the requester and a dirty transfer to the directory while transitioning to "Invalid". Furthermore, the cache can also receive a writeback request from the processor in this state, where it has to forward the request to the directory and transition to "Waiting for Writeback Response."

If the cache receives a simple writeback exclusive ack in "Waiting for Writeback Response," the cache can transition to "Invalid" without any other operations. If the cache receives an intervention, it would transition to "Waiting for Writeback Busy Ack," where it would wait for a writeback busy ack before transitioning to "Invalid". Similarly, if the cache receives a writeback busy ack first, it would transition to "Waiting for Intervention" and wait for the intervention before it can transition to "Invalid". It is also possible that the directory is in one of the busy states while the cache is "Waiting for Writeback Response," and the cache has to resend its writeback request if that happens.

6 Experimental Setup

In this section, I will go over the experimental setup used for running the simulations and explain why these parameters were chosen. The processor is modeled as superscalar and out-of-order to simulate the R10000 processor that is used in the SGI Origin 2000 computer systems. The rest of the components have their parameters configured according to Table 8. The number of cycles that each of the components take are approximated from the latency values in the SGI Origin 2000. The values for the network are approximated from the remote memory latencies in the SGI Origin system, and the values used for main memory are interpreted from the Origin's local memory latencies. It is assumed that writes take longer than reads and misses take longer than hits.

Benchmark	Command Line Parameters
synthetic benchmark	-n500
fft	-p[processor count] -m12 -n65536 -l4
radix	-p[processor count] -n262144 -r1024 -m524288
ocean	-p[processor count] -n258 -e1e-7 -r20000.0 -t28800.0
cholesky	-p[processor count] -B32 -c16384 $<$ tk15.O

Table 9: Benchmark parameters

The network is modeled according to the equation in Figure 11. The Network Latency (T_L) is determined by a combination of the Network Minimum (T_m), Network Random Range (T_R), Network Multiplier (T_X), and the number of messages in the network (N). Network Minimum is a lower bound to how long the message is in the network. Messages will be in the network for at least this time before being sent to the receiver. It is chosen to be close to the remote memory latencies in the SGI Origin system. Random Network Range is used to introduce some randomness into the network model. This is chosen to be 10% of the Network Minimum so that there would be some variation in the network latency, but not be large enough for the random values to affect the overall latency significantly. The Network Multiplier is used as part of the calculation for the additional time it would take for a message to pass through the network when there are other messages in the network. The dependence of the network latency on network traffic uses a linear model, which means that the more messages there are in the network, the higher the network latency is. The model takes into account additional messages in the network by multiplying Network Multiplier with the number of messages in the network to get the additional network delay. The Network Multiplier is chosen so that it has noticeable difference on the simulation compared to a model that does not take the network load into account; however, it is chosen to be small enough that it does not completely dominate the network latency. So the network latency for a single message is Network Minimum plus a random number between 0 and Network Random Range plus the Network Multiplier times the number of messages in the network.

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 results in unnecessary network traffic. We perform a check to see if the destination for a message is the same as the source of the message before sending the network message. If this is the case, it just forwards it to the local hub and the message does not get into the network.

Interfacing with the network are many nodes, which contain a cache, a directory, and a portion of the main memory that forms part of the DSM. The latencies and block sizes for the caches are configured to resemble the latencies on the R10000. For example, L1 cache is 64KB because this is the size of the L1 cache in the R10000 (32KB instruction + 32KB data). The L2 cache in has a size of 512KB, since that is the minimum that the SGI Origin 2000 can support, and we want the L2 cache size to be small to incur more messages in the network, since the point of interest is on the protocols and their difference in generating messages from requests. The parameters for the memory are significantly larger than the caches, but is still relatively small to simulate the behavior of main memory in a DSM[3].

All of the SPLASH-2 benchmarks, unless otherwise specified, are run with 64KB L1 cache and 512KB L2 cache, with the processor count between 4 and 32. The number of processors were chosen to show the difference between various number of processors and their effect on the protocol. All of the benchmarks are run with one thread on each node, which means that there are as many threads as there are processors.

7 Results

In this section, I will discuss the results of running the simulation on the SESC simulator. The parameters used in all benchmarks are summarized in Table 9. The individual sections regarding these benchmarks will go more in depth into these parameters. 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

```
int myInt = 1

readInt(numberOfLoops):
    loop numberOfLoops times:
        barrier1
        barrier2
        read myInt

incInt(numberOfLoops):
    loop numberOfLoops times:
        barrier1
        increment myInt
        barrier2

main(numberOfLoops):
    spawn thread incInt(numberOfLoops)
    spawn thread readInt(numberOfLoops)
```

Figure 12: Read increment synthetic benchmark pseudocode

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 SPLASH-2 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. First, I will go over a small synthetic benchmark that exploits the differences in protocol. Then, I will present the results of running the simulator on the SPLASH-2 benchmarks.

To show the differences between these two protocols, I show their performances according to several benchmarks. The Origin protocol is superior than BIP because BIP uses more messages than the Origin protocol for several different types of requests. To demonstrate this, I created a synthetic benchmark that generates many requests, which will generate different amount of messages between Origin protocol and BIP. I will also show the protocols' performance on real-world benchmarks. This will demonstrate their performance when faced with benchmarks that are not specifically geared to exploit the differences between the protocols. The experiments I ran will indicate that the difference in protocol is what causes the difference in performance, and not because of some other differences in the simulator.

7.1 Synthetic Benchmark

This section introduces the synthetic benchmark as well as the results received from running the synthetic benchmark. The synthetic benchmark was run on a four processor system and was created to illustrate the strength of the Origin protocol. With this synthetic benchmark, I am trying to force a node to repeatedly obtain access to a block and then be forced to give it up as some other node requests it. The easiest way to achieve this is by creating situations that involve many evictions forced by the directory. In other words, I want to show that in situations where the directory has to transition in or out of "Exclusive," and the directory uses intervention messages, that the required number of network transactions is higher on BIP than on Origin.

The pseudocode for the synthetic benchmark is shown in Figure 12. The global int, myInt, is the variable that will be read and written in this synthetic benchmark. The readInt and the incInt functions both take in a parameter, numberOfLoops, that tells the function how many times to read the myInt variable. The difference is that the readInt function only reads from myInt and the incInt function reads myInt and also increments it. The main function takes one argument, NumberOfLoops. NumberOfLoops specify how many times the threads would read the variable myInt. We want to avoid the scenario where readInt execute all its iterations before incInt executes its iterations, so we use barriers to make sure that the threads are synchronized. Here, I provide a description of the strict order of accesses afforded by the barriers. After readInt and incInt both hit barrier1, incInt would increment myInt, and readInt would hit barrier2. After incInt hits barrier2, readInt would be free to access myInt. Meanwhile, the loop in incInt would hit barrier1, preventing it from accessing myInt until the loop in readInt has finished accessing myInt. After the loop in

$$homenode = \frac{address}{blocksize} mod \ numberOfProcessors$$

Figure 13: Formula for calculating home node from address

```
int myInt = 1

readInt(numberOfLoops):
    loop numberOfLoops times:
        barrier1
        barrier2
        read myInt

main(numberOfLoops):
    spawn thread readInt(numberOfLoops)
    spawn thread readInt(numberOfLoops)
```

Figure 14: Read-only synthetic benchmark pseudocode

readInt hits barrier1 again, the cycle would repeat. The combination of barrier1 and barrier2 makes sure that the functions will always follow this order: incInt, readInt, incInt, readInt...

We also want to make sure that the home node where myInt resides, the readInt processor, and the incInt processor are all on separate nodes. From the address of the myInt variable, we can figure out the home node using the formula in Figure 13. As long as we have the address of the shared variable, we can calculate the home node, since we set the block size and the number of processors. In the system used for running the synthetic benchmark, the block size was 64 bytes and the number of processors is 4. After getting the home node, we spawn threads readInt and incInt to make sure that they reside on different processors than the home node and from each other. This will ensure that I get the behavior I want from my simulations, since the protocols are used whenever the nodes request information from each other, and not if the request is local (i.e., between a local cache and a local directory).

In order to make sure that the overheads of setting up the barrier are not counted towards the final number of messages, I also wrote a variation of the same program that calls only readInt but not incInt. This program is shown in Figure 14. Instead of the main process spawning a readInt thread and an incInt thread, this program will spawn two readInt threads onto two different processors. The barriers are not truly necessary in this case, since the order of the reads do not matter. Once the processors obtain the initial shared read, the two threads would be able to read myInt without sending more messages. This "base case" program will tell us the execution time and number of messages that are used to set up the barrier. We will take the results of the program in Figure 12 and subtract the number of messages and the execution time of the base case program to find the execution time and number of messages that I am trying to measure.

I will also explain the number of messages I get based on the protocol. Table 10 is the table for the messages sent for each increment in BIP. The requester would have held this block in "Shared," since the previous request into the directory involved a shared read request. Initially, the requester would send an exclusive read to the directory and transition from "Shared" to "Waiting for Exclusive Read Reply." Next, the directory sends an intervention exclusive request to a sharer S and transitions from "Shared" to "Waiting to Send Invalidates" when it receives the exclusive read. Sharer S is chosen by the directory to be a random sharer that is not the requester. Since there are only two processor in this example, we know that sharer S can only be the node that is not the requester. When sharer S receives the intervention exclusive request, it would transition from "Shared" to "Shared Waiting for Exclusive Ack" and send an eviction message to the directory. After the directory receives the eviction message, it would transition from "Waiting to Send Invalidates" to "Waiting for K Invalidates, J Invalidates Received So Far", send an invalidate message to the requester, and send an eviction ack to sharer S. When sharer S receives the eviction ack, it would transition from "Shared Waiting for Exclusive Ack" to "Invalid." After the requester receives an invalidate in "Waiting for Exclusive Read Reply," it would send an invalidate ack to the directory, but it does not transition into a different state. When the directory receives the invalidate ack, it would transition from "Waiting for K Invalidates, J Invalidates Received So Far" into "Exclusive," and send an exclusive reply to the requester. Finally, the requester receives the exclusive reply and transitions from "Waiting for Exclusive Read Reply"

Time Step	Node	State Transition	Messages Emitted
1	Requester	Shared⇒Waiting for Exclusive Read Reply	exclusive read→Directory
2	Directory	Shared⇒Waiting to Send Invalidates	intervention exclusive→Sharer S
3	Sharer S	Shared⇒Shared Waiting for Exclusive Ack	eviction→Directory
4	Directory	Waiting to Send Invalidates⇒Waiting for K Invalidates, J Invalidates Received So Far	invalidate→Requester eviction ack→Sharer S
5	Sharer S	Shared Waiting for Exclusive Ack⇒Invalid	None
6	Requester	None	invalidate ack→Directory
7	Directory	Waiting for K Invalidates, J Invalidates Received So Far⇒Exclusive	exclusive reply→Requester
8	Requester	Waiting for Exclusive Read Reply⇒Dirty Exclusive	None

Table 10: Messages generated for each increment in BIP

Time Step	Node	State Transition	Messages Emitted
1	Requester	Shared⇒Waiting for Exclusive Read Response	exclusive read→Directory
2	Directory	Shared⇒Shared Exclusive (Memory-Access)⇒Exclusive	exclusive reply with invalidates pending→Requester invalidate→Previous Owner
3	Requester	Waiting for Exclusive Read Response⇒Waiting for K Invalidates, J Invalidates Received So Far	None
4	Previous Owner	Shared⇒Invalid	invalidate ack→Requester
5	Requester	Waiting for K Invalidates, J Invalidates Received So Far⇒Exclusive	None

Table 11: Messages generated for each increment in Origin

to “Dirty Exclusive” without sending any more messages into the network. In total, seven messages get sent on the network.

The Origin protocol has a different behavior for each increment, which is shown in Table 11. Initially, the requester sends an exclusive read to the directory while transitioning from “Shared” to “Waiting for Exclusive Read Response.” Since the previous request into the directory involved a shared read request, the requester would have held this block in “Shared.” When the directory receives the exclusive read, it transitions from “Shared” to “Shared Exclusive (Memory-Access)” to “Exclusive” while sending out an exclusive reply to the requester and an invalidate to the previous owner. When the requester receives the exclusive reply with invalidates pending, it transitions from “Waiting for Exclusive Read Response” to “Waiting for K Invalidates, J Invalidates Received So Far,” but it does not emit any messages. When the previous owner receives the invalidate, it would transition from “Shared” to “Invalid,” and send an invalidate ack to the requester. Finally, the requester receives the invalidate ack and transitions from “Waiting for K Invalidates, J Invalidates Received So Far” into “Exclusive.” Therefore, four messages are sent into the network.

The number of messages that has to be generated for each read in BIP is shown in Table 12. Initially, the requester will send a shared read request message to the directory, and transition from “Invalid” to “Waiting for Shared Read Reply.” Next, the directory will receive the shared read request, and transition from “Exclusive” to “Exclusive Shared Waiting for Reply,” sending an intervention shared request to the previous owner during the transition. Then, depending on whether or not the previous owner has the block in “Clean Exclusive” or “Dirty Exclusive,” the previous owner would send a shared transfer or a shared writeback to the directory, then transition to “Shared.” The directory would receive the shared transfer/writeback, then transition from “Exclusive Shared Waiting for Reply” into “Shared,” and send out a shared reply to the

Time Step	Node	State Transition	Messages Emitted
1	Requester	Invalid⇒Waiting for Shared Read Reply	shared read→Directory
2	Directory	Exclusive⇒Exclusive Shared Waiting for Reply	intervention shared→Previous Owner
3a	Previous Owner	Clean Exclusive⇒Shared	shared transfer→Directory
3b	Previous Owner	Dirty Exclusive⇒Shared	shared writeback→Directory
4	Directory	Exclusive Shared Waiting for Reply⇒Shared	shared reply→Requester
5	Requester	Waiting for Shared Read Reply⇒Shared	None

Table 12: Messages generated for each read in BIP

Time Step	Node	State Transition	Messages Emitted
1	Requester	Invalid⇒Waiting for Shared Read Response	shared read→Directory
2	Directory	Exclusive⇒Busy-Shared (Memory-Access)⇒Busy-Shared	intervention shared→Previous Owner speculative reply→Requester
3a	Previous Owner	Clean Exclusive⇒Shared	shared ack→Requester shared transfer→Directory
3b	Previous Owner	Dirty Exclusive⇒Shared	shared response→Requester shared writeback→Directory
4	Directory	Busy-Shared⇒Shared	None
5a	Requester	Waiting for Shared Read Response⇒Shared Waiting for Speculative Reply⇒Shared	None
5b	Requester	Waiting for Shared Read Response⇒Waiting for Shared Response/Ack⇒Shared	None

Table 13: Messages generated for each read in Origin

original requester. Finally, the requester receives the shared reply and transitions from "Waiting for Shared Read Reply" to "Shared." In summary, four messages get send for the read in BIP.

For a read in Origin, the messages sent in the system is shown in Table 13. First, the requester would send the shared read to the directory and transition from "Invalid" to "Waiting for Shared Read Response." Next, the directory receives the shared read and transitions from "Exclusive" to "Busy-Shared (Memory-Access)," then from that state to "Busy-Shared." No messages are emitted on the network for the transition from "Exclusive" to "Busy-Shared (Memory-Access)." During these transitions, the directory sends an intervention shared request to the previous owner and a speculative reply to the requester. If the previous owner receives the intervention shared request in the "Clean Exclusive" state, it would send a shared ack to the requester and a shared transfer to the directory, while transitioning to "Shared." However, if the previous owner receives the message in "Dirty Exclusive," it would send a shared response to the requester and a shared writeback to the directory while transitioning to "Shared." When the directory receives the shared transfer/writeback, it would transition from "Busy-Shared" to "Shared" without sending any messages into the network. Finally, the requester will receive both the speculative reply and the shared response/ack, transitioning from "Waiting for Shared Read Response" to "Shared" through either of the paths shown in time step 5a or 5b. The requester would not send any more messages into the network. This operation takes five messages to perform. The overall number of messages sent into the network for each read/increment is shown in Table 14. This shows that the Origin protocol should use two fewer messages for each read/increment function call pair.

The synthetic benchmark was run with a parameter of "-n500" to simulate reading the myInt variable 500

	BIP	Origin
Read	4	5
Increment	7	4
Total	11	9

Table 14: Number of messages sent into the network for each read/increment

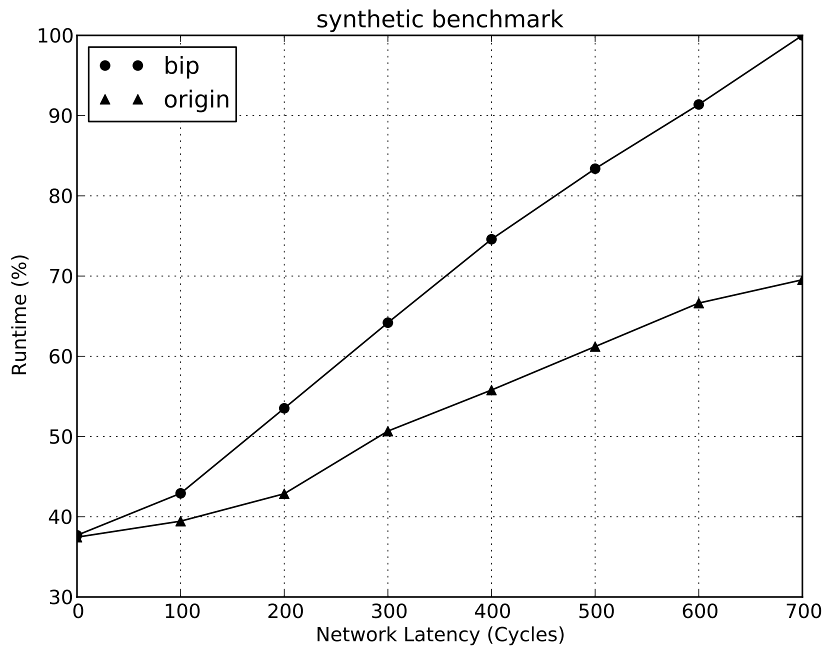


Figure 15: Synthetic benchmark results - execution time versus network latency

times and incrementing it 500 times in the two processors. Since we have to make sure that the home node is different than the readInt processor and the incInt processor and processor numbers can only be in powers of 2, this simulation was run on a system with 4 processors. I vary the Minimum Network variable in the network latency equation from Figure 11 in order to see what effects varying the network latency has on the execution time. The results show the difference in execution time by varying this parameter between 0 and 700 in increments of 100. Since both threads are using the same memory location, this program generates a large number of invalidates. This forces the block that contains myInt to get continuously evicted, which will highlight the major benefit of the Origin protocol versus BIP. The advantage of Origin protocol over BIP allows it to use less messages. This will also test the effectiveness of the Origin protocol in reducing overall runtime by allowing the data node to reply directly to the requesting node, without having to send the message first to the directory.

The results of running this benchmark are presented in Figure 15 and Table 15. The results in the graph represent the execution time after subtracting the execution time of the "base case" that only reads myInt. The "net number of network messages" also represent the number of messages after subtracting the overhead of setting up barriers in the "base case." Even after subtracting the overhead, we can see that BIP uses more messages to complete the benchmark versus the Origin protocol. This would indicate that the required number of transactions for BIP is higher than Origin protocol. We also find that running the simulation with a network latency of 0 achieves results with very similar execution time for BIP versus Origin protocol. This demonstrates that the difference in runtime between the two protocols is dependent on the latency in the network, and not on other components in the system. If we increase the network latency by varying the

Protocol	Number of Network Messages in Read-Only Benchmark	Number of Network Messages in Read Increment Benchmark	Net Number of Network Messages
BIP	59	4824	4765
Origin	64	4132	4068

Table 15: Synthetic benchmark results - number of network messages

Minimum Network variable, we can achieve a more realistic network model. Higher network latency allows the difference between the two protocols to show. We can see from the graph that the Origin protocol is faster than BIP in each of the cases when we increase the network latency to above 0. Also apparent is the gap between BIP and Origin, which increases as the network latency increases. This means that the increase in performance from using the Origin protocol versus using BIP becomes more significant as we increase the network latency.

7.2 SPLASH-2 benchmarks

The SPLASH-2 benchmarks represent a set of programs designed to facilitate the study of systems like the one mentioned in this report[4]. The results from running these benchmarks will be used to show the performance of BIP versus Origin protocol. An execution time ("runtime") versus processor count graph is shown for each of these benchmarks. In each of these graphs, the execution time is normalized. For example, in each graph featuring the runtime, the results will not be shown in raw runtime numbers; the numbers represent the current y value versus the highest y value on the graph. A number of messages versus processor count graph and an L2 cache misses versus processor count graph for each of these benchmarks are presented in a similar fashion. The benchmarks that will be presented from the SPLASH-2 Benchmark Suite are FFT, Radix, Ocean, and Cholesky.

7.2.1 FFT

This benchmark was run with "-p[processor count] -m12 -n65536 -l4" as the parameter, and it was run from the beginning to the end. The fourth graph on "runtime versus network latency" was run using 16 processors. These configurations mean that the benchmarks were run with 2^{12} total complex data points transformed, 65536 number of cache lines, and $Log_2 4$ of cache line length in bytes. The total complex data points transformed is increased from the default shown in the fft header file so that it would show a difference between different number of processor counts. The results of running the FFT benchmarks are shown in figure Figure 16. We see that for both Origin and BIP, 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. As we can see, the biggest speedup is going from 4 processor to 8 processors. This is apparent in both protocols. This could be due to inherently non-parallelizable code in the benchmark, leading diminishing returns when we move to higher processor counts. To demonstrate this, the higher the number of cores we use, the more messages we have in the system, which means that more messages are contributing to the network congestion, slowing the whole system down. We know this because the misses from the L2 cache is the same for both Origin protocol and BIP across all processor counts. We see from varying the Minimum Network variable in network latency that higher network latencies contribute to higher runtimes for both protocols, but what is interesting is that the gap between BIP and Origin become bigger as the network latency increases, which would indicate that the differences between the two protocols is in the network.

7.2.2 Radix

This benchmark was run with "-p[processor count] -n262144 -r1024 -m524288" as the parameter, and it was run from the beginning to the end. These parameters translate to 262144 number of keys to sort, with a 1024 radix for sorting, and a maximum key value of 524288. These were the default configurations from the radix header file. The results from running the Radix benchmark is shown in Figure 17. These results are

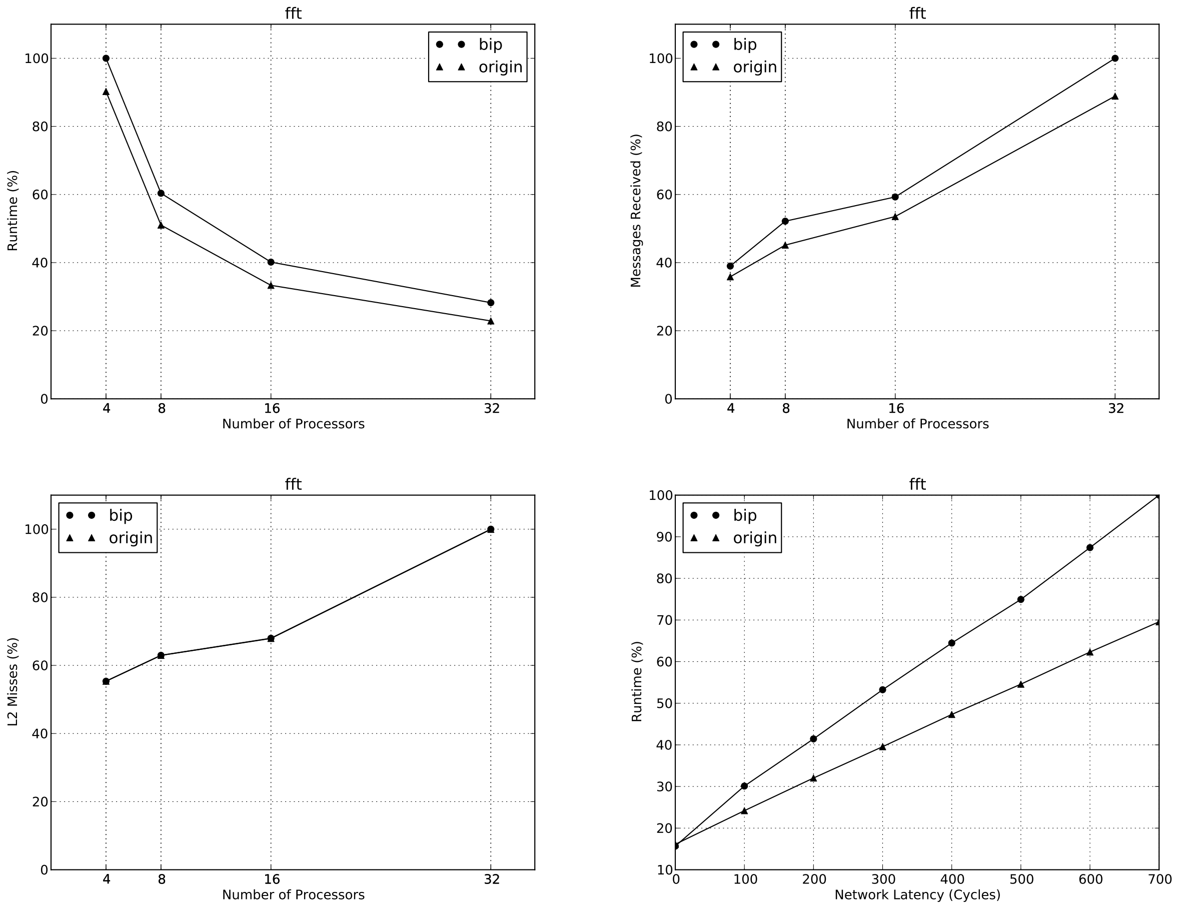


Figure 16: FFT results

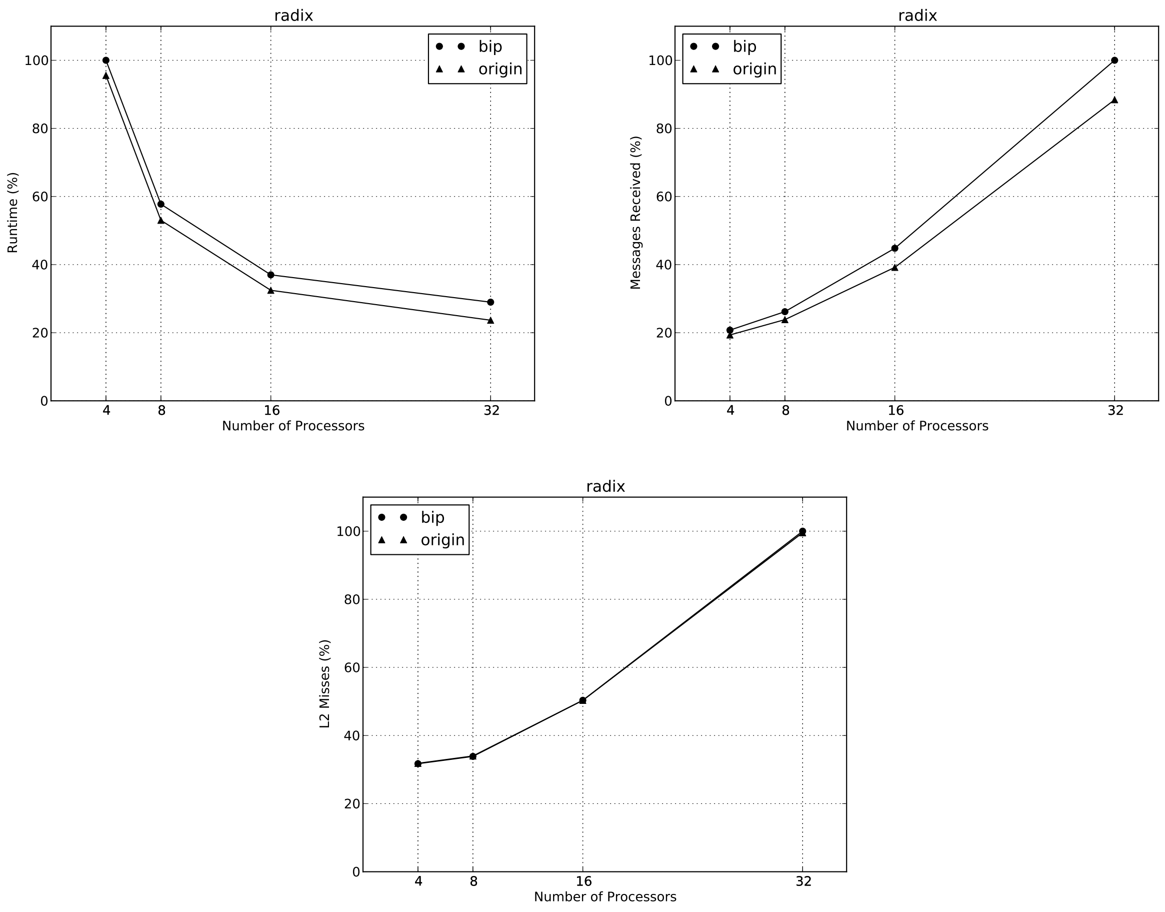


Figure 17: Radix results

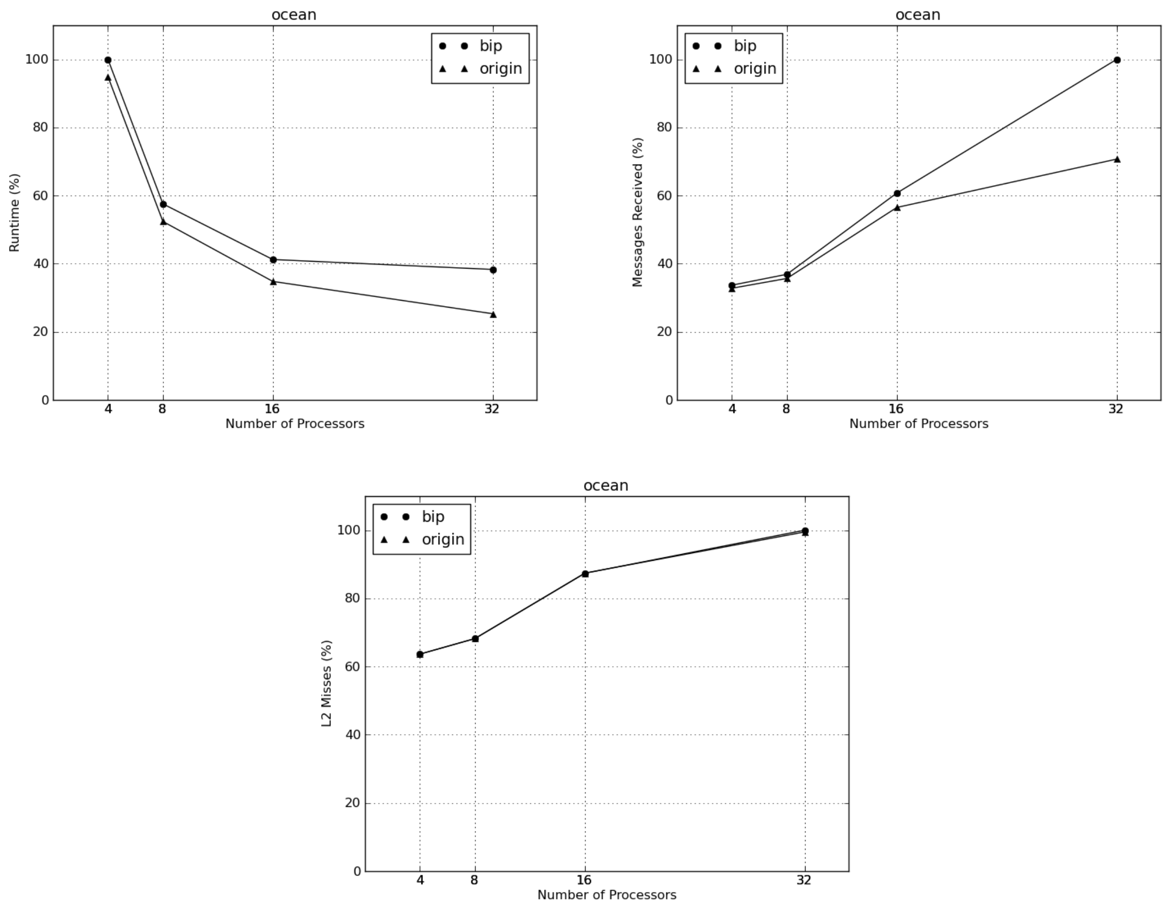
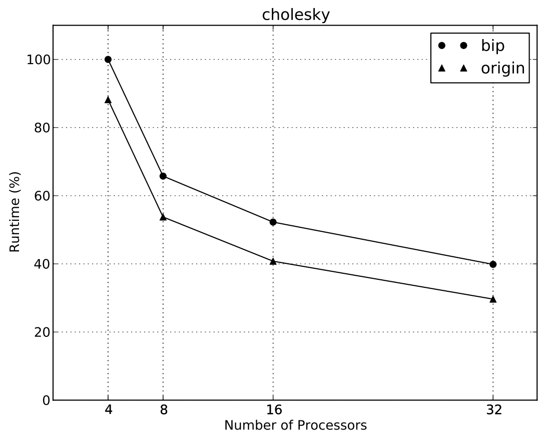


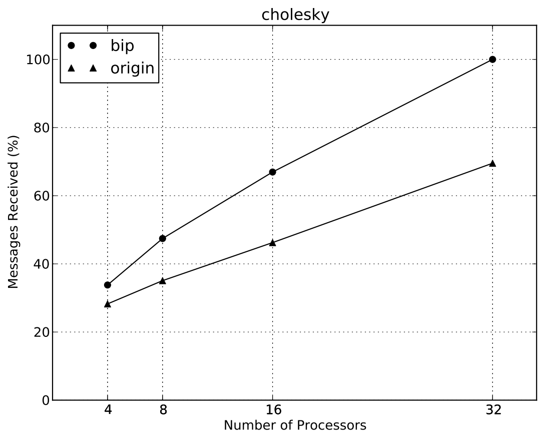
Figure 18: Ocean results

similar to the FFT benchmark for the total runtime. Again, the biggest speedup is going from 4 processor to 8 processors even though there is noticeable speedup throughout the system. However, in either case, the Origin protocol is faster than BIP. This is because BIP generated more messages, as the L2 cache for Origin protocol and BIP have the same number of misses. The higher number of messages in the system when processor count increases contributes to the slowdown in the system, as evidenced by the non-ideal increase in speedup across both protocols.

7.2.3 Ocean

This benchmark was run from the beginning to the end using "-p[processor count] -n258 -e1e-7 -r20000.0 -t28800.0" as the parameter. These parameters simulate a 258x258 ocean, an error tolerance for iterative relaxation of 1×10^{-7}, a 20000 meter between grid points, and a timestep of 28800 seconds. These are the default configurations for the ocean benchmark. The results for the Ocean benchmark is shown in Figure 18. For both benchmarks, the greatest speedup occur when going from 4 to 8 processors. However, the interesting part is that in the results for 32 processors, the Origin protocol is noticeably faster than BIP, when compared to the speedup experienced in 4, 8, or 16 processors. This is because the Origin protocol sends proportionally fewer messages than BIP at 32 processors compared to the results received at other processor counts. This allows the Origin protocol to experience fewer slowdowns related to traffic congestion in the system. Again, we see that the number of misses are equal, which means that the speedup of the Origin protocol over BIP can be attributed to the higher number of messages sent in BIP.





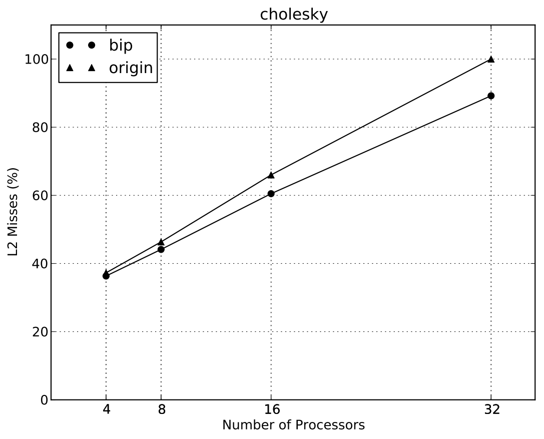


Figure 19: Cholesky results

7.2.4 Cholesky

This benchmark was run with the parameter "-p[processor count] -B32 -c16384 < tk15.O" from the beginning to the end. This indicates that the simulation used a postpass partition size of 32 and a cache size of 16384 bytes, with the tk15 input file. These follow the recommended default configurations for the Cholesky benchmark. The results for Cholesky is shown in Figure 19. Like the other benchmarks, the runtime decreases for both protocols as the processor count increases. The Origin protocol caused more misses in the L2 cache. Despite this, BIP still sent more messages into the network than Origin protocol. The difference in their total runtime can be attributed to Origin's reduction in messages sent as compared to BIP.

8 Problems and Future Works

There were some problems in implementing the directory-based cache coherence protocol in the SESC simulator. 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. In addition, there are edge cases in the protocol, not specified by the Origin paper, where it is necessary to guess what the specific implementation would be. The other problem is that, like any simulator, this is a model of the real system, so the results might not exactly match what can happen on the real system.

As for future work, 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[1][5].

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.

9 Conclusion

We have arrived at the conclusion that the Origin-based directory protocol is superior in many situations compared to BIP. Origin-based systems achieve lower runtime based on their ability to inject fewer messages into the network.

References

[1] James Laudon and Daniel Lenoski. The sgi origin: A ccnuma highly scalable server. pages 241–251. Proceedings of the 24th Annual International Symposium on Computer Architecture, May 1997.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 500 Sansome Street, Suite 400, San Francisco, CA, 4th edition, 2007.

[3] L. Gwennap. MIPS R10000 uses decoupled architecture. *Microprocessor Report*, 8(14):18–22, 1994.

[4] Evan Torrie Jaswinder Pal Singh Steven Cameron Woo, Moriyoshi Ohara and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. pages 24–36. Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 1995.

[5] Keith I. Farkas and Norman P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. pages 211–222. Proceedings of the 21st International Symposium on Computer Architecture, April 1994.