Lab Assignment 5: Coherence

1 Objective

The goal of this lab is to enhance your understanding of cache coherence. In the lectures you learned the fundamental concepts of coherence protocols. However, even simple three-state protocols, such as MSI (Modified-Shared-Invalid), are more complex in real life implementations. This lab will introduce you to these more complex aspects of coherence protocols such as transient states. This lab is quite challenging. As always, you are encouraged to start early.

At the end of this lab:

- You will have a thorough understanding of how coherence protocols are implemented and why we need transient states.
- You will have implemented an MSI coherence protocol in the SLICC domain-specific language in the gem5 simulator and have validated its correctness.

All work on this assignment should be done in groups of two.

The remaining of the handout is organized as follows: Section 2 provides an overview of coherence and introduces you to transient states and other coherence implementation challenges. Section 3 introduces you to the gem5 simulator and the SLICC domain specific language via a walk-through example. Section 4 defines the problem you are asked to solve, while Section 5 provides useful guidelines for code development and debugging. Section 6 specifies the prelab preparation you should complete. Sections 7 and 8 describe the assignment deliverables and a high-level overview of the marking scheme respectively. Finally, the Appendix provides a handy reference of all the provided SLICC actions.

2 Cache Coherence

2.1 Coherence - An overview

Multi-core systems with a shared-memory model need to provide a unified view of each memory location to all executing contexts. Such a coherent system is what we all intuitively expect, but this is not enforced by the architecture without a coherence protocol.

Let us look at an example. Think of a modern computer system with two cores with shared memory, with private write-back L1 data caches. This can very well be your laptop with a Core-i3 processor. Let us assume that a single program (two threads) is running on both cores. Thread 1 reads memory location A from memory, with an initial value of 10. The contents of memory location A now reside in Core's 1 private L1 data cache. Some instructions later, the same thread writes 20 to memory location A. The second thread, which is running on Core 2, wants to read memory location A. Since it does not have address A cached, it sends a read request to memory. In a system without any coherence protocol that second thread will read a stale value of 10.

To provide a coherent view of memory, and avoid examples such as the one above, coherence protocols enforce two invariants [3]:

- the *Single-Writer, Multiple-Reader (SWMR)* Invariant, and
- the *Data-Value* Invariant

The first invariant specifies that at any given point in time a memory location can either be read by multiple (one to all) cores of a system, or it can be written by a single core. Write permissions also imply read permissions, i.e., the core that can write to a block can also read it. The aforementioned example violated this first principle. The second invariant guarantees that data modifications are properly propagated. For example, if multiple cores read a block at the same time, then they should all read the same value.

2.2 Implementing a Coherence Protocol

Cache coherence is enforced via a set of coherence controllers. Coherence controllers are finite-state machines (FSMs) associated with the hardware storage structures of a system (e.g., caches, memory). The coherence protocol specification includes:

- the possible FSM states
- the permitted state transitions
- the events that can trigger a transition, and
- the actions that take place at a state transition

Coherence is commonly maintained at the granularity of cache blocks. Therefore, for each block that is stored somewhere in the system, a coherence controller keeps track of its current coherence state.

In principle, a coherence controller receives a tuple {Current State S, Event E} as an input, and generates a new tuple {Action Set A, NextState N} as output. The block then transitions to coherence state N and a set of actions A execute. It is possible that states S and N are the same, and that action set A is empty.

The events which can cause a protocol transition are:

- core requests that involve a memory access (e.g., loads, stores, instruction fetches)
- write-back requests (i.e, when a cache evicts a local copy of a block)
- data transfers (e.g., a reply from memory on a load miss), and
- coherence requests (e.g., invalidations, acknowledgements)

The last category of events allows the various coherence controllers to communicate with each other.

2.2.1 Directory Protocols

There are two main classes of protocols: *Directory-Based* and *Snoop-Based* protocols. In this assignment we will focus on the former. The main characteristic of directory protocols is the presence of a logically centralized structure (i.e., the directory) that acts as the ordering point of all memory requests. The directory can be co-located either with the last-level shared cache of the system, or with the memory controller. In all cases, the directory controller orchestrates the interactions between the various cache coherence controllers.

In the earlier example, the directory would receive Core 2's read request for block A. The directory controller, knowing that block A has been modified by Core 1, would issue an invalidation message to Core 1. Eventually, Core 2 would get the block in shared state, along with its latest data value.

The state information for each block is explicitly stored in the directory. However, each cache should also store state information for its cached blocks. For example, if Core 0 has a block cached in Modified state in its L1-D cache, then it can safely read/write to it without contacting the directory. To facilitate this, we add a few extra bits alongside the tag of a cache block, to specify its protocol state. As protocols have a small number of stable states, the storage overhead is minimal. The next section discusses transient states.

2.2.2 Transient States

In the protocol state diagrams, state transitions along with their associated actions happen "instantaneously", or rather in an **atomic fashion**. Unfortunately, in real hardware this atomicity requirement is violated.

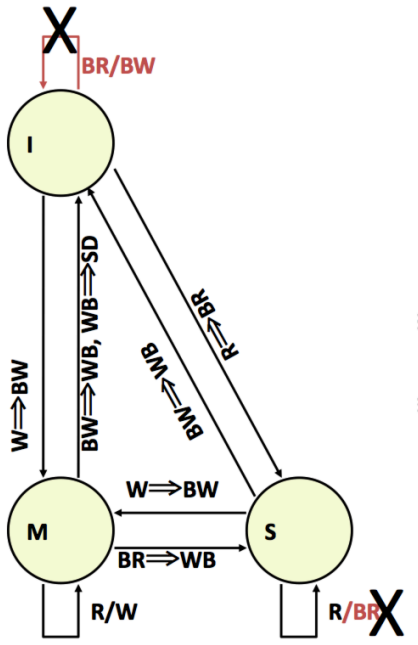

Figure 1: MSI Cache Controller Transition Diagram (from Lecture Slides)

Let us look at an example, assuming an MSI directory protocol with two cores. Core 1 executes a read request for block B. The L1 cache controller performs the cache lookup and finds block B in invalid state. According to the MSI transition diagram in Figure 1, Core 1 should perform a set of actions A and transition to shared state. The set of actions A is (1) issue a read request to the directory, and (2) wait for a data reply.

The set of actions A cannot complete atomically. Action A1 will have to traverse the interconnection network and experience delays before it will reach the directory. Even then, it will probably wait in some incoming queue before it gets processed. Therefore there is a window from the time the request got issued, until the time it reached the directory and got serviced. During this time window, another request (e.g., a write request from Core 2) might have already arrived at the directory and been serviced. If Core 1 had transitioned to S state immediately, then the system would have violated the single writer/multiple readers invariant from Section 2.1.

Transient states resolve this issue. Transient states are intermediate states that facilitate the transitions between two stable states. In the previous example, a transient state IS_D would denote

that a transition from invalid state (I) to shared state (S) is in process, and the coherence controller is waiting for a data reply (D). Once the data reply arrives, the controller can safely transition to stable state S. In the meantime, any local read/write request to this block will be stalled, as the block has pending coherence requests and does yet have read/write permissions.

The number of transient states depends greatly on the coherence protocol, its implementation, and the underlying system. This is contrary to the number of stable states, which are the same irrespective of the coherence protocol (e.g., snoop-based versus directory-based).

Blocks in transient states are stored outside the cache, in some temporary buffers commonly known as Miss Handling Registers (MSHRs). A cache-entry is also reserved at this point. Each MSHR entry holds the data block, along with its current transient state information. Core-initiated accesses to blocks in transient states will stall. Once the block transitions to a stable state, the data block and stable state information is copied to the cache entry, and any previously stalled accesses can safely proceed.

There are different ways to stall a request, all with their own trade-offs. The most common option is to stall the request at the head of the incoming FIFO queue, thus blocking all requests buffered after it. This option, although simple, might unnecessarily delay independent requests.

2.2.3 Deadlock and Virtual Networks

Directory coherence protocol implementations are also susceptible to deadlock. Deadlock is the phenomenon where no forward progress is made in a system because of two inter-dependent events. For example, if an event A can only occur after event B completes, and event B can only complete after event A occurs, then none of these two events will ever take place. This cyclic-dependence between A and B indefinitely deadlocks the system.

Here is an example of such a cyclic-dependence in an MSI directory protocol: Let us assume a two-core system, and a block A in modified state is in Core 2's cache. Core 1 issues a load request for block A, which misses in its local cache. Thus, the coherence controller sends a read request to the directory. The directory forwards this read request to the owner of the block (i.e. Core 2) asking it to forward the dirty data block to Core 1. At this point, Core 1 issues a store request to block A. However, as this block is in transient state pending the data reply for the previous load, the controller stalls this request. Eventually, Core 2 receives the forwarded request from the directory and sends the data block as a reply to Core 1. The reply gets buffered in the single incoming queue of Core 1, behind the store request for the same block. A deadlock situation just occurred. The store request will be stalled until the data reply gets processed, and the data reply cannot be processed unless it reaches the head of the queue (i.e., unless the stalled store request proceeds).

The aforementioned deadlock scenario can be avoided if separate resources (i.e., queues, networks) are used for different message types. For example, if the data response from Core 2 was placed in a separate queue than Cores 1's local store request, then no deadlock would occur. **Virtual Networks** is the commonly used solution. Virtual networks are logically separate networks with their own set of incoming/outgoing queues and buffers. These networks are not physically separate (i.e., the interconnection links and wires are not replicated). Messages are "tagged" with the id of their virtual network, and can thus bypass messages travelling on other virtual networks, and be allocated in the proper queues.

For the purpose of an MSI protocol, three virtual networks are sufficient to avoid cyclic resource dependences [3]. Specifically, separate virtual networks are required for:

- Cache-Initiated Requests (e.g., read/write requests, and write-backs)

- Directory-Initiated or Forwarded Coherence Requests (e.g., forwarded requests to a remote cache, invalidations, write-back acknowledgements)
- Cache Responses (e.g., data replies, invalidation acknowledgements)

These virtual networks have already been configured in the simulator and you do not need to modify them in your implementation.

2.3 Further Reading

This section presented an overview of coherence and introduced you to some real-implementation challenges of coherence protocols. To reinforce these new concepts, you are required to read Section 8.2 (pages 153-161) of the **Primer on Memory Consistency and Coherence** book [3]. Focus on Figure 8.3 which offers a high-level overview of the MSI protocol, and Tables 8.1 and 8.2 which provide a detailed protocol specification in the presence of transient states. You will use these tables as a reference point for your MSI protocol implementation.

The **Primer on Memory Consistency and Coherence** book, part of the Synthesis Lectures on Computer Architecture, is an excellent resource on all things coherence. An electronic version of this book is available from the University of Toronto libraries website via this link: `https://doi-org.myaccess.library.utoronto.ca/10.2200/S00962ED2V01Y201910CAC049`. Chapters that might be of interest are Chapter 2 which covers the basics of cache coherence, Chapter 6 which provides a high-level overview of coherence protocols, and Chapter 8 which focuses on Directory coherence protocols. Finally, for the curious reader, Chapter 9 delves into some more advanced topics of cache coherence, beyond the scope of this course.

3 GEM5 - A walk-through example

For the purpose of this lab, you will use **gem5**, a full-system architectural simulator [1, 2]. Gem5 supports a wide set of Instruction Set Architectures (ISAs), CPU and memory models. For this assignment, we will make use of its coherence modeling capabilities.

You can checkout a tar version of a modified gem5 simulator from the course's home directory on the ug machines as follows:

```
cp /cad2/ece552f/gem5.tgz ~/Lab5/
cd ~/Lab5/
tar -xvf gem5.tgz
```

3.1 Building gem5

Due to some libraries and package dependencies, you have to add two directories to your path so you can build the simulator. You also need to ensure that the SWIG environment variable is set and that you have properly configured your terminal. To achieve all these, add the following lines to your ∼/.bashrc file (assuming you are using the bash shell):

```
#Paths to dependencies
PATH=/cad2/ece552f/gem5_dependencies/bin:$PATH
LD_LIBRARY_PATH=/cad2/ece552f/gem5_dependencies/lib:$LD_LIBRARY_PATH

export PATH
export LD_LIBRARY_PATH
export SWIG=/cad2/ece552f/gem5_dependencies/bin/swig
```

```
#Set your terminal to xterm so you do not get $ signs in the paths when building gem5
export TERM=xterm
```

Remember to source your bashrc file after the modifications.

```
. ~/.bashrc
```

To build the simulator run the following **case-sensitive** command from the root of your Gem5 tree:

```
scons PROTOCOL=MSI CPU_MODELS=TimingSimpleCPU build/ALPHA/gem5.opt
```

SCons is a build tool, a substitute of the Make utility, gem5 uses. In the build command, we specify the coherence protocol we want to build the simulator for (MSI), the ISA (ALPHA), and the CPU model (TimingSimpleCPU). Finally, we also specify the executable configuration. In this case we request the optimized version (i.e., gem5.opt), which maintains basic debugging functionality such as assertions and debugging print statements. Other options of interest are: gem5.debug where optimizations are turned off, and gem5.fast where debug functionality is removed [6].

Building the system for the first time will take a while; it takes approximately 15 minutes on ug170. So once you see that it has passed the initial checks, it might be a good idea to take a break. While scons is running, it is creating and populating a build directory. The final size of the build directory is approximately 544MB. Please make sure you have sufficient disk space, as there is a disk quota on the ug machines. You can check the disk usage of a given directory via this command: *du -hsc DIRNAME*.

3.2 SLICC

The focus of the development for the current lab will be the **gem5/src/mem/protocol/** directory. Specifically, you will only need to modify the following two files:

- gem5/src/mem/protocol/MSI-cache.sm
- gem5/src/mem/protocol/MSI-dir.sm

Do not modify any other files!

Gem5 uses SLICC, a domain specific language, to define the coherence protocol. Remember that a coherence protocol implementation consists of (a) a set of coherence controllers (i.e., FSMs) and (b) a set of interactions between these controllers [3]. The file *MSI.slicc* contains a list of all the files used by the MSI protocol; the protocol's name is specified in the first line.

```
protocol "MSI";
include "RubySlicc_interfaces.slicc";
include "MSI-msg.sm";
include "MSI-cache.sm";
include "MSI-dir.sm";
```

The *RubySlicc_interfaces.slicc* file contains all the files relevant to Ruby memory system. You will not need to modify this file, so treat it as an included header file. The *MSI-cache.sm* file holds the definition of the L1 Cache coherence controller, whereas the *MSI-dir.sm* file defines the FSM associated with the directory controller. In addition to core-initiated memory requests, I/O devices can also access memory via DMA requests. You can safely ignore these for your implementation.

The various controller types communicate via different classes of Memory Messages, as specified in the *MSI-msg.sm* file. For example, the L1 Cache controller sends a *RequestMsg* message to the Directory Controller, whereas the directory replies to the requesting core with a *ResponseMsg* class type. Different structure definitions are needed, as some data fields are not shared across different message classes. For example, a data-less response message from the directory should inform the requestor of the number of pending acknowledgements.

The fields of a RequestMsg are presented below as a reference. Some fields, such as address, message size, are shared across all message classes.

```
// RequestMsg (and also forwarded requests)
structure(RequestMsg, desc="...", interface="NetworkMessage") {
  Address Address,               desc="Physical address for this request";
  CoherenceRequestType Type,     desc="Type of request (GetS, GetM, PutM, etc)";
  MachineID Requestor,           desc="Node who initiated the request";
  NetDest Destination,           desc="Multicast destination mask";
  DataBlock DataBlk,             desc="data for the cache line";
  MessageSizeType MessageSize,   desc="size category of the message";
}
```

All messages of the same class "share" a virtual network to avoid deadlock situations. As you might expect, there are different types of Request Messages within a given class. The Type field specifies the message type. All permitted message types for a given class are defined in an enumeration block in the same *MSI-msg.sm* file. In addition, generic request and ruby request types are defined in *RubySlicc_Exports.sm*. For example, the core will issue an IFETCH message to read an instruction from memory, versus a LD message to read data from memory. Table 1 lists all message types of interest, along with a brief explanation. The virtual channel numbers are the ones presented later in Figure 2.

Core-Initiated Requests (mandatoryQueue)		
LD	RubyRequestType	Load
IFETCH	RubyRequestType	Instruction Fetch
ST	RubyRequestType	Store
Cache Controller Initiated Requests (Virtual Channel 2)		
GETM	CoherenceRequestType	Get Modified (write request)
GETS	CoherenceRequestType	Get Shared (read request)
PUTM	CoherenceRequestType	Put Modified (writeback of modified block)
PUTS	CoherenceRequestType	Put Shared (writeback of shared block)
Coherence Responses (Virtual Channel 4)		
DATA_FROM_OWNER	CoherenceResponseType	Data from remote Owner (L1)
DATA_FROM_DIR	CoherenceResponseType	Data from Directory, along with Ack Count
DATA	CoherenceResponseType	Data from Cache to Directory
ACK	CoherenceResponseType	Invalidation Acknowledgement
Forwarded/Directory Initiated Coherence Requests (Virtual Channel 3)		
WB_ACK	CoherenceRequestType	Writeback Acknowledgement
INV	CoherenceRequestType	Invalidation
GETS	CoherenceRequestType	Forwarded GETS
GETM	CoherenceRequestType	Forwarded GETM
Memory-Related Requests (memQueue)		
MEMORY_READ	MemoryRequestType	Read data block from memory
MEMORY_WB	MemoryRequestType	Write data block to memory

Table 1: Message Types (Message Type, Message Class, Explanation)

3.3 Cache Coherence Controller - A walk-through example

Let us walk through the MSI-cache.sm file and its explicit sections. A cache controller is defined as an instance of SLICC's machine datatype. The machine has some data members associated with it, such as its nearest storage structure (e.g., L1 cache for this instance). In addition, some access latencies, and a few configuration parameters are associated with this entry. Finally, the Sequencer is the component which feeds the memory hierarchy with core-initiated requests, i.e., the RubyRequestType messages presented earlier in Table 1.

```
machine(L1Cache, "MSI Example L1 Cache")
: Sequencer * sequencer,
  CacheMemory * cacheMemory, //ECE552: Unified L1 Instruction and Data Cache
  int cache_response_latency = 12,
  int issue_latency = 2,
  bool send_evictions = true
```

Each state-machine communicates with other components in the system via message buffers. Each message buffer effectively acts either as an output port (e.g., sending requests from the local L1 cache to the network) or as an input port (e.g., receiving coherence responses from the network). An example is provided for reference below:

```
MessageBuffer requestFromCache, network="To", virtual_network="2", ordered="true", vnet_type="request";
MessageBuffer responseToCache, network="From", virtual_network="4", ordered="true", vnet_type="response";
```

You will notice that a message buffer connects to a specific type of virtual network. Messages travel on different virtual networks to avoid deadlock. In the aforementioned example, cache requests "travel" on virtual network 2, whereas responses on virtual network 4.

The next logical section in the controller definition is the state declaration (e.g., line 39 in MSI-cache.sm). All permitted states, both stable and transient, need to be declared here with the following info: state name, access permissions, and short description. This section will be filled-in by you. For the sake of an example, let us assume we have a protocol with stable states A and B.

```
A, AccessPermission:Invalid, desc="Stable state A, block is invalid";
B, AccessPermission:Read_Only, desc="Stable state B, block is shared";
AB_D, AccessPermission:Busy, desc="Issued read request from state A, waiting for data";
```

Access-Permissions are specified in RubySlicc_Exports.sm file. Valid data can have (a) Read_Only, or (b) Read_Write permissions. Invalid data can be (a) Invalid, (b) NotPresent, or (c) Busy. For the purpose of this assignment you can use Invalid and NotPresent interchangeably. Also, a block with "Busy" permissions is a block in a transient state. As it was mentioned earlier, these blocks are not allocated in the cache, but rather in a Miss Status Handling Register (MSHR) or a Transient Block Entry (TBE in SLICC notation) since they have pending actions.

Transient states that occur during a transition from state X to state Y, waiting for event W should be represented as follows: XY_W.

The FSM transitions between states based on specific events. The permitted set of events for each cache controller is declared via an enumeration block. Each event has a specific name and a descriptor. For example, the enumeration below declares a load event and a Fwd_GetS event.

```
 enumeration(Event, desc="Cache events") {
    Load,       desc="Load request from processor";
    Fwd_GetS,   desc="GetS forwarded from the directory";
}
```

Each coherence controller has some structures associated with it. Cache entries are of type Entry, while transient blocks are stored in TBEs. Both types of entries hold a state identifier (stable or transient) and the Data Block associated with this address. A TBE entry also keeps track of the number of pending acknowledgements.

In addition to the network buffers, a cache controller also has a mandatory queue buffer. This mandatory queue buffers requests from the local core towards the local L1 cache.

The cache controller reacts to specific messages from its incoming queues. In the beginning, when all queues are empty, the first queue that receives messages is the mandatory Queue. If a queue is not empty (i.e., function isReady() returns true), then we "peek" at the incoming queue. The peek function grabs the message at the head of the queue, without dequeuing it, and returns it in the *in_msg* variable. If-else statements map the incoming message type to an event and trigger the appropriate transition. Remember that each incoming queue services different message types, to avoid cyclic dependencies and deadlock.

The code snippet from the mandatory-Queue of the cache controller is below:

```
in_port(mandatoryQueue_in, RubyRequest, mandatoryQueue, desc="...") {
  if (mandatoryQueue_in.isReady()) {
    peek(mandatoryQueue_in, RubyRequest, block_on="LineAddress") {

      Entry cache_entry := getCacheEntry(in_msg.LineAddress);
      if (is_invalid(cache_entry) &&
          cacheMemory.cacheAvail(in_msg.LineAddress) == false ) {
        // make room for the block
        trigger(Event:Replacement, cacheMemory.cacheProbe(in_msg.LineAddress),
                getCacheEntry(cacheMemory.cacheProbe(in_msg.LineAddress)),
                TBEs[cacheMemory.cacheProbe(in_msg.LineAddress)]);
      }
      else {
        trigger(mandatory_request_type_to_event(in_msg.Type), in_msg.LineAddress,
                cache_entry, TBEs[in_msg.LineAddress]);
      }
    }
  }
}
```

Transitions map a tuple {State, Event} to a potentially new State. In the following example of an MI protocol, a block in Invalid State transitions to a transient IM_D state in the event of a load. Before this transition completes, a set of actions take place, in the specified order. Be careful, that the set of actions should never be empty. The lack of actions means you are stalling the request so you should call the *z_stall* action.

```
transition(I, Load, IM_D) {
   v_allocateTBE;
   //.... more actions here...
 }
```

If an event does not change the current state of a block, then the transition function receives only two arguments. Sometimes it is possible that a set of initial states (e.g., A and B) transition to the same next state N in case of an event E. You can declare this as follows:

```
transition({A, B}, E, N) {
  //...actions go here...
}
```

The same holds for a set of events that behave in the same manner (e.g., a Load and an Ifetch request).

```
transition(I, {Load, Ifetch}, IM_D) {
  //...actions go here...
}
```

All actions specified in the transition functions have already been declared for you. An action is characterized by a keyword (e.g., v_allocateTBE), an abbreviation (e.g., "v"), and a descriptor. The body of the action implements the necessary functionality.

```
action(v_allocateTBE, "v", desc="Allocate TBE") {
    TBEs.allocate(address);
    set_tbe(TBEs[address]);
  }
```

The GEM5 website provides a similar walk-through example, along with some additional information [5]. Remember that after any modification to the MSI protocol, you have to rebuild the simulator.

3.4 Protocol Validation via Ruby Random Tester

In order to validate the correctness of your protocol you can run the following from your gem5 directory:

```
./build/ALPHA/gem5.opt ./configs/example/ruby_random_test.py
```

The ruby_random_test.py python script receives a set of configuration parameters. You can see all of them via -help. The most important ones are [4]:

```
-n, --num-cpus  Number of cpus injecting load/store requests to the memory system.
--num-dirs      Number of directory controllers in the system.
-m, --maxtick   Number of cycles to simulate.
-l, --checks    Number of loads to be performed.
--random_seed  Seed for initialization of the random number generator.
```

The random tester injects random requests to the L1 caches. If a bug in your code exists the test will not complete, but will instead exit with an assertion or an error message. To better facilitate debugging you can use the **ProtocolTrace** debug flag, which generates a complete dump of all the protocol transitions and actions of the coherence controllers in your system.

```
./build/ALPHA/gem5.opt --debug-flags=ProtocolTrace ./configs/example/ruby_random_test.py
```

You can also add debug messages (i.e., DPRINTF) statements in your code, enabled via the **RubySlicc** debug flag. Some have already been provided for you. For example the following message prints the address and type of an incoming message.

```
DPRINTF(RubySlicc, "L1 Cache Ctrl - Message type %s for addr. %s\n", in_msg.Address, in_msg.Type);
```

4 Problem Statement

In this assignment you are asked to specify all the states and transitions for an MSI protocol. You can only modify the files:

- gem5/src/mem/protocol/MSI-cache.sm
- gem5/src/mem/protocol/MSI-dir.sm

All required actions, functions, message buffers, etc have already been specified for you.

Figure 2 shows a high-level overview of the system with all the coherence controllers and their input/output ports. The numbers on the arrows denote the virtual network associated with each port.

The system for which you are writing the MSI system has the following characteristics:

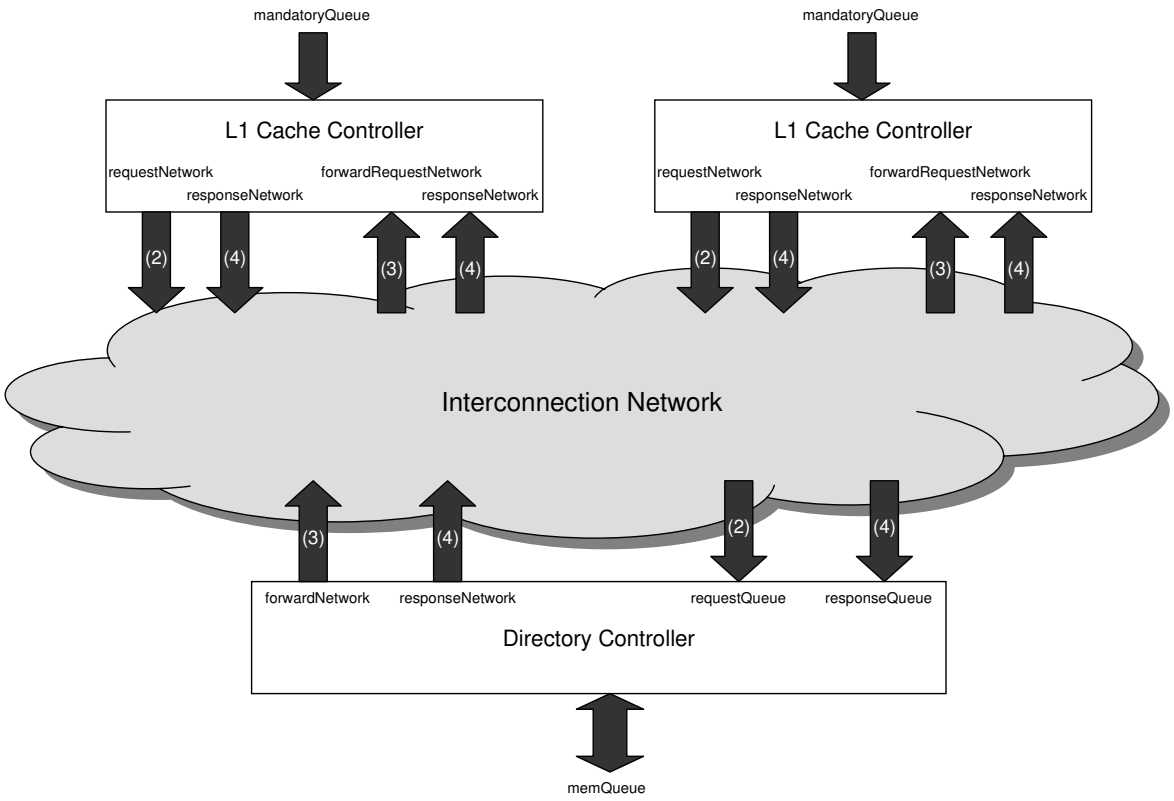

Figure 2: An overview of the system

- Private, unified, L1-Caches (i.e., both instruction fetches and memory requests access the same cache). Each L1 cache is 256B, 2-way set-associative, with 64 Bytes cache blocks. This cache size is unrealistically small to stress the coherence controllers and identify bugs/deadlocks sooner.
- A single directory controller, not co-located with the memory controller. The state of blocks in the directory is cache-centric, i.e., it reflects the state of all copies of this block in the L1 caches. Each directory entry keeps track of the following:
  1. Block state in the directory; can be stable or transient.
  2. A list of all the sharers of this block. This list is empty if the block is modified or invalid.
  3. The current owner of this block, if modified. This field is empty if the block is shared or invalid.
  4. The data block associated with this address. The data block is up-to-date, if the block is shared, and thus the directory can reply with the data. If the block is modified, then the directory's copy is out-of-date, and the block's owner provides the data reply.
- The MSI is an invalidation-based protocol.
- No silent evictions of clean blocks are permitted for the L1 caches. A *silent eviction* is when a cache does not issue a write-back request when it replaces a clean block; instead it "drops it on the floor". In this system, caches always issue write-backs when they replace blocks. Thus the directory always has an up-to-date list of all the sharers of a block.
- The presence of data blocks in the directory entries makes this implementation slightly different than the one in the Coherence Primer book. Specifically, it introduces more transient states in Table 8.2 [3]. There are two main implications:

1. First, when a block is in invalid state in the directory, the directory controller needs to retrieve the data block from memory. Therefore, you have to explicitly issue read requests to memory and properly wait for the memory reply.
2. Second, writeback of dirty data should be properly propagated to the directory entries. You also have to issue an explicit memory writeback request, and properly wait for the memory write-back acknowledgement.

5 Methodology

The first step in your methodology should be to craft the necessary transient states for an MSI protocol. You should use Tables 8.1 and 8.2 of the Primer on Coherence book [3] as a guideline. Ask yourselves why a particular state exists (i.e., a table row), what are the permitted events for a given state (i.e., a table column), and which actions should take place (i.e., what should be the contents of a given cell).

Here are some preparation questions to help you in this process. Succinct answers to these questions should be part of your final report.

1. How does the FSM know it has received the last Inv Ack reply for a TBE entry?
2. How is it possible to receive a PUTM request from a NonOwner core? Please provide a set of events that will lead to this scenario.
3. Why do we need to differentiate between a PUTS and a PUTS-Last request?
4. How is it possible to receive a PUTS-Last request for a block in modified state in the directory? Please provide a set of events that will make this possible.
5. Why is it not possible to get an Invalidation request for a cache block in a Modified state? Please explain.
6. Why is it not possible for the cache controller to get a Fwd-GetS request for a block in SI_A state? Please explain.
7. Was your verification testing exhaustive? How can you ensure that the random tester has exercised all possible transitions (i.e., all {State, Event} pairs)?

5.1 Invalid Transitions

Answering all these questions, might not be as intuitive at first. Iteratively add the states and transitions for the scenarios you feel comfortable about. The first time you run your code, you are destined to see a fatal error of this form:

```
fatal: Invalid transition
system.l1_cntrl0 time: 63835 addr: [0x53c0, line 0x53c0] event: Load state: I
```

The second line tells you that you got a Load event for a block in state I, and that you have no transition to handle this. This error occurred for the "l1_cntrlX" controller, where X is the core id starting from zero.

If you did not add a transition on purpose, because you did not think a {State, Event} situation would be possible, then more debugging is required. The fatal error tells you the offending address. Rerun your code with ProtocolTrace and RubySlicc debug flags enabled, redirecting your output to a file. Search this file for the line address (e.g., 0x53c0) and walk-through all the transitions that have happened so far. This way you can understand why such a transition is possible.

5.2 Actions on Transitions

A transition is always associated with a set of actions. This set should never be empty in SLICC. All actions have already been declared for you. You need to find the correct set of actions, and their order, for each transition.

Here are some guidelines:

- Think if a transition should result in a stable or transient state.
- Think if this transition should cause a coherence message or a reply to be sent.
- Remember to pop a message from its incoming queue, once you finish its processing.
- Order your actions appropriately. For example, if you are processing a message, the action which pops it from the queue should be last.
- Inform the core (i.e., the sequencer) upon request completion.
- Make sure you deallocate an Invalid cache block.

Feel free to add debugging (DPRINTF) statements to the various actions.

We do not expect that you will need to define any new actions in your protocol implementation. Need for a new action may indicate you have made an error in your design. However, if you are confident in your implementation, you are free to add actions to the files. Also, due to slight implementation variations, you may not need to use all the provided actions. Please notice that some actions are quite similar, and have only subtle differences (e.g., operate on different incoming queues). Make sure you select the proper action, based on your current state and triggering event. Use the Appendix at the end of this handout as a reference guide for actions.

5.3 Deadlock

The random ruby tester has a deadlock detection mechanism embedded in it. A "deadlock detected" error is triggered when a processor has not made any progress in a fixed number (a few thousands) of cycles. To debug this, run the random tester with debug flags enabled. Search for the last transition performed by the deadlocked processor. Monitor all protocol stalls near that time, to determine all the event types which are pending for this controller. For example, if all stalls are for blocks in IS_D state, then the issue is with the directory not sending a data reply. Iteratively repeat this process until you can determine the offending address and identify your code bug.

5.4 Other Mistakes

- A " *panic: Action/check failure: proc: 3: address: ..*" error message shows some inconsistency in the DataBlk field of a message. Make sure you correctly propagate the data information along with the address.
- An *isReady fails()* message is an indicator that you are peeking an empty queue. Make sure you peek the appropriate message buffer for the virtual network of the current request type.

5.5 Rigorous Testing

Once you do not get any errors for the single CPU configuration, increase the number of CPUs in your system to two (-n 2 configuration option) Run the tester for different load counts up to one million [4], as specified via the -l configuration parameter. Repeat this process for different core counts up to 16 processors, iteratively fixing all the bugs you find along the way. Remember to rebuild gem5 after every modification to the MSI protocol.

5.6 Table View

SLICC gives you the option to generate a table view of your protocol implementation, similar to that found in Table 8.1 and 8.2 [3]. This option is disabled by default. To enable, edit the file SConsopts in src/mem/protocol, and change the line:

```
opt = BoolVariable('SLICC_HTML', 'Create HTML files', False)
```

to:

```
opt = BoolVariable('SLICC_HTML', 'Create HTML files', True)
```

The html table can then by viewed by opening build/ALPHA/mem/protocol/index.html in your browser (you will need to recompile). The columns correspond to the different cache/directory events, and each row corresponds to a state or transient state. Each cell in the table tells you the actions that are performed when a particular event happens during a particular state. In addition, some cells will indicate to which state the FSM will transition to. Clicking on an action or event will reveal its short description in the bottom status bar frame. Clicking on a state will highlight the cells which transition to that state in blue. This table can be very useful for visually identifying problems in your protocol.

6 Prelab

The prelab is worth 1/6 of the overall lab mark. Please complete the following steps before coming to the lab:

- Thoroughly read and understand all necessary background. First, focus on Sections 2 and 3 of this handout and then read Tables 8.1 and 8.2 [3].
- Write most of the state and transitions code. You are **strongly** encouraged to have your code pass the uniprocessor random test.
- Answer the following questions:
  1. Why are transient states necessary?
  2. Why does a coherence protocol use stalls?
  3. What is deadlock and how can we avoid it?
  4. What is the functionality of Put-Ack (i.e., WB-Ack) messages? They are used as a response to which message types?
  5. What determines who is the sender of a data reply to the requesting core? Which are the possible options?
  6. What is the difference between a Put-Ack and an Inv-Ack message?
- Be prepared to answer any high-level questions about coherence, transient states, deadlock, and your code.

7 Lab Deliverables

At the end of this assignment you should submit your code modifications along with a short report. The necessary commands are listed below:

```
//Command to submit your files
submitece552f 5 MSI-cache.sm MSI-dir.sm report.pdf

//Command to view your submitted files (i.e., sanity check)
submitece552f -l 5
```

Remember to adhere to the required naming scheme; misnamed files will not be marked. All code modifications should be within the explicitly marked sections in the MSI-cache.sm and MSI-dir.sm files. Do not forget to comment your code! And make sure not to leave the Unix permissions to your code open!

The report should contain:

1. Answers to the Prelab questions from Section 6.
2. Answers to all the questions from Section 5.
3. A small table documenting all your protocol modifications (e.g., new transient states) with respect to Tables 8.1 and 8.2 [3].
4. A brief statement of work completed by each partner.

The length of the report should not exceed four pages.

8 Due Date and Marking Scheme

This assignment is due on **Wednesday December 6, 2023 at 6:00pm**. It is worth **6**% of the total course mark. The pre-lab will constitute 1/6 of the overall mark.

The correctness of your code will be tested with an auto-marker and a series of tests. To guarantee fairness, and have deterministic tests, we will use a predefined random-seed for the ruby-tester. For every test your code passes, you will get partial marks.

For example, you will receive partial marks if your code successfully completes the single CPU test. Passing the multiprocessor test with larger caches, where very few replacements occur, will also give you partial marks. The ultimate goal is to be able to run your code with very small caches, multiple CPUs (e.g., up to 16), and around a million loads, bug and deadlock-free.

You can use the following list of potential tests as a guideline. Please note that this is not an exhaustive list of all the auto-marker's tests.

```
./build/ALPHA/gem5.opt ./configs/example/ruby_random_test.py -n 4
./build/ALPHA/gem5.opt ./configs/example/ruby_random_test.py -n 2 -l 10000
./build/ALPHA/gem5.opt ./configs/example/ruby_random_test.py -n 16 -l 1000000
./build/ALPHA/gem5.opt ./configs/example/ruby_random_test.py -n 2 --l1d_size 8kB --l1d_assoc 4
```

9 Questions

A list of frequently asked questions has been posted in the Lab Material section of Quercus. Please refer to this list first to see if your question has already been answered. If you post a question that the TA feels has already been answered in this document, they will refer you to the handout.

Please post clarifications questions on Piazza. Please give your questions on Piazza descriptive titles to help other students identify if their question has already been asked/answered. Titling your question: "Lab 5 question" is not helpful. Please DO NOT post code snippets; this constitutes cheating and you will be penalized. For a code-specific question send a private message to a TA.

Good Luck!

References

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. SIGARCH Comput. Archit. News 39, 2 (August 2011), 1-7.

[2] Jason Lowe-Power et al. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152.

[3] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood. 2020. A primer on memory consistency and cache coherence, 2nd ed., ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers. Accessible from University of Toronto libraries via `https://doi-org.myaccess.library.utoronto.ca/10.2200/S00962ED2V01Y201910CAC049`.

[4] `https://www.gem5.org/documentation/general_docs/debugging_and_testing/directed_testers/ruby_random_tester`

[5] `https://www.gem5.org/documentation/general_docs/ruby/slicc`

[6] `https://www.gem5.org/documentation/general_docs/building`

A Appendix - Action Descriptions

To make code development easier, you can find here a list of all the available actions per coherence controller, along with a short description.

A.1 Actions for the Cache Controller

1. **as_issueGetS**: Issue a request for read-only access (i.e., GETS request type) to the directory controller. This action creates an outgoing message *out_msg* and properly populates its fields (e.g., address, requestor, destination, size etc). It finally enqueues this message to the requestNetwork_out output port.
2. **am_issueGetM**: Issue a request for read-write access (i.e., GETM request type) to the directory controller.
3. **bs_issuePUTS**: Send a PUTS message to the directory, along with data, upon local replacement of a shared block.
4. **bm_issuePUTM**: Send a PUTM message to the directory, along with data, upon local replacement of a modified block.
5. **e_sendDataFromCacheToRequestor**: Send a DATA_FROM_Owner reply to a forwarded incoming coherence request. The destination is a remote cache controller and the requested block is present in a stable state in the local cache.
6. **ee_sendDataFromTBEToRequestor**: Send a DATA_FROM_Owner reply to a forwarded incoming coherence request. The destination is a remote cache controller. However, unlike the previous action, the requested block is now present in a transient state in a TBE entry.
7. **de_sendDataFromCacheToDir**: Send a DATA reply message to the directory for a cached block.
8. **dee_sendDataFromTBEToDir**: Send a DATA reply message to the directory for a block in transient state.
9. **fi_sendInvAck**: Send a single invalidation acknowledgement message (ACK) to a remote cache controller.
10. **i_allocateL1CacheBlock**: Allocate a L1 cache block, if there is no valid cache entry for this address.
11. **h_deallocateL1CacheBlock**: Deallocate a L1 cache block upon cache replacement or invalidation.
12. **m_popMandatoryQueue**: Dequeue a request from the mandatory incoming queue. Messages are dequeued and handled in-order.
13. **n_popResponseQueue**: Dequeue a request from the responseNetwork queue.
14. **o_popForwardedRequestQueue**: Dequeue a request from the forwardRequestNetwork queue.
15. **p_profileMiss**: Call this action whenever a core-initiated request results in a cache miss (i.e., block is invalid or it does not have proper access permissions).

16. **r_load_hit**: Inform the local core that a load has completed. Call this action if the data block was present with read permissions in the local cache.
17. **rx_load_hit**: Inform the local core that a load has completed. Unlike *r_load_hit*, we had a cache miss. The cache got the block either from the directory or from a remote L1 cache.
18. **s_store_hit**: Inform the local core that a store has completed. Call this action if the data block was present with write permissions in the local cache.
19. **sx_store_hit**: Inform the local core that a store has completed. Unlike *s_store_hit*, the block was either not present in the local cache or was present but with read-only permissions. The cache got the block either from the directory or from the previous owner of that block (i.e., a remote L1 cache).
20. **u_writeDataToCache**: When a DATA reply is received, copy the data block to the previously allocated cache entry.
21. **q_updateAckCount**: Update the number of pending acknowledgements in a TBE entry. When a TBE entry is allocated, the field pendingAcks is set to 0. The in_msg.AckCount when received from the directory is a negative number (i.e., minus the number of sharers). Acknowledgement messages from remote cores, have a positive AckCount of one.
22. **forward_eviction_to_cpu**: Inform the core (i.e., the Sequencer in gem5 terminology) that a block was replaced or invalidated.
23. **v_allocateTBE**: Allocate a TBE entry. The block is in a transient coherence state.
24. **w_deallocateTBE**: Deallocate a TBE entry. The block is now in a stable coherence state and resides in the cache.
25. **x_copyDataFromCacheToTBE**: Copy a data block from a cache to a TBE entry. Remember that a block in transient state might still be asked to provide data to forwarded requests.
26. **z_stall**: Stall, i.e., do nothing.

A.2 Actions for the Directory Controller

1. **kk_removeRequestSharer**: Remove the requestor of this message from the sharers list of a directory entry.
2. **ars1_addRequestorToSharers**: Add message requestor to the sharers list of a directory entry. Action taken upon memory reply.
3. **ars2_addRequestorToSharers**: Add message requestor to the sharers list of a directory entry. Here the triggering factor is an incoming message from the requestQueue, while in the previous ars1_addRequestorToSharers action, the triggering factor is a message from the memQueue.
4. **aos_addOwnertoSharers**: Add the previous Owner of this block, to the sharers list of this directory entry.
5. **a_sendWriteBackAck**: Send a writeback acknowledgement (WB_ACK) to a cache, as a reply to a PUT message. The triggering factor is a message received in the requestQueue port.
6. **la_sendWriteBackAck**: Send a writeback acknowledgement (WB_ACK) to a cache, as a reply to a PUT message. Unlike the previous action, here the triggering factor is a write-back acknowledgement from memory.
7. **c_clearOwner**: Clear the owner field of a directory entry.
8. **cs_clearSharers**: Clear the sharers fields for a directory entry.
9. **d_sendData**: Send a DATA_FROM_DIR message to a remote cache along with a zero AckCount. The directory was in invalid state, and the data block was retrieved from memory (i.e., received from memQueue).

10. **ds_sendSharedData**: Send a DATA_FROM_DIR message to a remote cache for a shared block. No invalidation acknowledgements from remote caches are required.
11. **ds2_sendSharedData**: Send a DATA_FROM_DIR message to a remote cache for a shared block. Invalidation acknowledgements from remote caches (i.e., previous sharers of this block) are required. The exact number is passed along with the reply in the AckCount field. The AckCount field can be zero, only if the requestor was the sole sharer for that block.
12. **e_ownerIsRequestor**: Set the requesting core as the new owner of a modified block.
13. **f_forwardRequest**: The directory forwards an incoming request (e.g., GETS, GETM) to the Owner of this directory entry.
14. **i_popIncomingRequestQueue**: Dequeue a request from the requestQueue.
15. **pr_popIncomingResponseQueue**: Dequeue a request from the responseQueue.
16. **qm_popMemQueue**: Deque a request from the incoming memory queue (memQueue).
17. **pl_writeDataToDirectory**: Copy incoming Data Block, from a write-back request, to the directory entry.
18. **pl2_writeDataToDirectory**: Copy incoming Data Block to the directory entry. This action gets triggered on a coherence response and not on a coherence PUTS/PUTM request as the pl_writeDataToDirectory action.
19. **vv_allocateTBEFromRequestNet**: Allocate a TBE entry.
20. **w_deallocateTBE**: Deallocate a TBE entry.
21. **z_stall**: Stall.
22. **qf_queueMemoryFetchRequest**: Send a MEMORY_READ request to memory (i.e., memQueue_out port). This action is necessary to retrieve the data for an invalid directory entry.
23. **lq_queueMemoryWBRequest**: Send a MEMORY_WB request for message received in requestQueue_in port.
24. **lq2_queueMemoryWBRequest**: Send a MEMORY_WB request for message received in responseQueue_in port.
25. **qwt_queueMemoryWBRequest_partialTBE**: Send a partial MEMORY_WB request for a message received in requestQueue_in port and with a TBE entry.
26. **w_writeDataToDirectoryFromTBE**: Copy data block from a TBE to a directory entry.
27. **fwm_sendFwdInvToSharersMinusRequestor**: Send Invalidations to all sharers of a directory entry, except for the requesting core.