CIS 540 Spring 2016 Homework 4, Due March 28
Modeling and Verification of Cache Coherence Protocol

The project involves modeling, simulation, and model checking in the popular software SPIN (see spinroot.com for all the details about the tool). The tool xspin has an easy-to-use interface (click on GettingStarted for instructions to run a demo of xspin). You need to get familiar with the tool by browsing through the manual, tutorial, and examples.

Here are a few notes to relate it to what we have discussed in the course.

Modeling

The modeling language is called Promela. The model consists of processes executing asynchronously and communicating by sending/receiving messages on channels. Thus, it is very similar to Chapter 4. You will have to learn the syntax though. The directory Test contains many examples.

Simulation

You can either simulate or verify the model. In simulation mode, a single execution is generated, and illustrated using a format called Message Sequence Charts, a commonly used standardized notation from UML.

Specifying Requirements

You can insert assert statements within the model: if an assertion fails this is reported as an error. You can also write LTL formulas. Read leader.ltl for sample specifications. SPIN compiles these formulas into what it calls never claims, which are exactly the monitors in Section 5.2.1 (you can write them directly as Promela models also.

Model Checking

The basic verification algorithm in SPIN is enumerative depth-first-search. A large number of optimizations have been implemented to make it scale to complex models. Performance of the verifier crucially depends on setting of parameters.

Project Description

After learning the basics of the tool, you need to model and analyze the cache coherence protocol described below. Most of the background information here is borrowed from [1].

Cache coherence protocols

Many modern computer systems have multi-core processors. Usually each processor core has its own cache and all cores can read from or write to a single shared memory. Having multiple copies of the same data in different caches may lead to inconsistency because each processor may change the data in its own cache. For example if multiple caches have access to the same data and one of them updates it, then the rest of the caches will see an outdated version of data. The goal of cache coherence protocols is to ensure that multiple cached copies of data are kept up-to-date.

In this project we will implement a simple cache coherence protocol. Our simplified system model includes a set of cores with their corresponding caches and cache controllers and a directory and its memory, as illustrated in Figure 1.

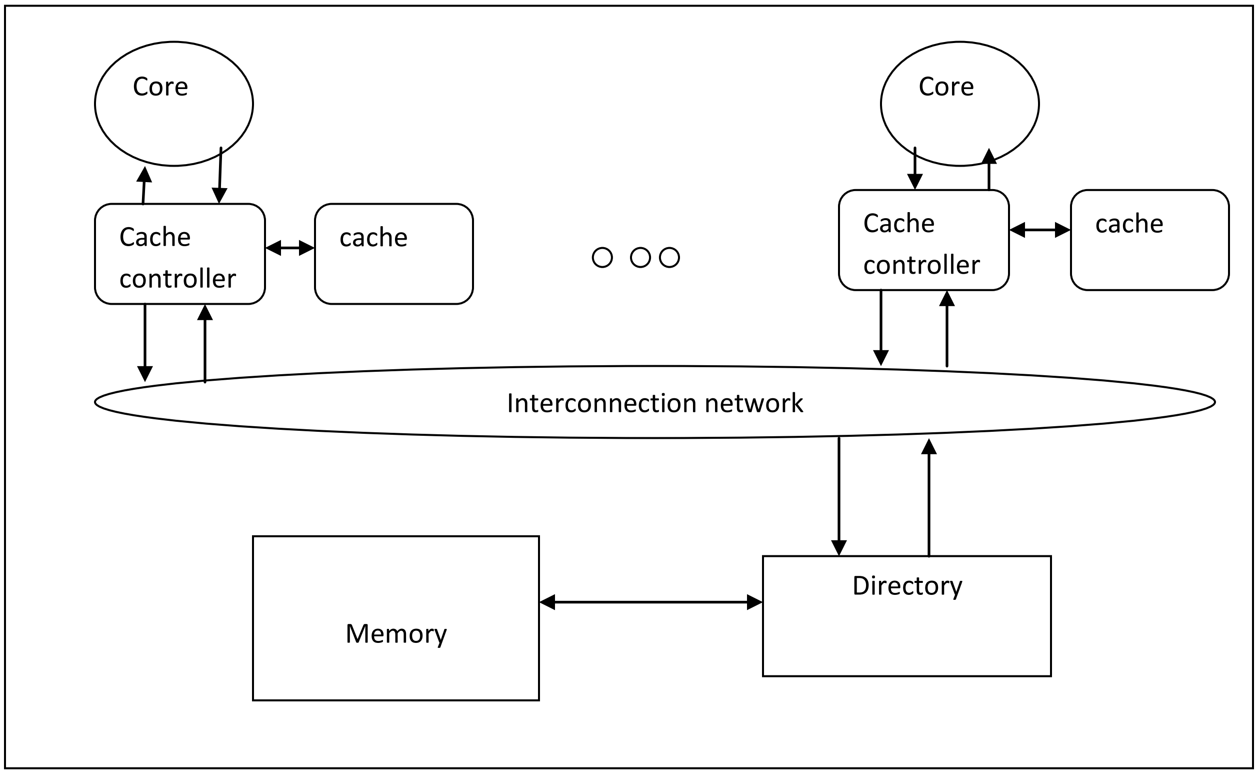

Figure 1: System Model

The basis of our preferred definition of coherence is the single-writer–multiple-reader (SWMR) invariant. For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it. Thus, there is never a time when a given memory location may be written by one core and simultaneously either read or written by any other cores.

In addition to the SWMR invariant, coherence requires that the value of a given memory location is propagated correctly. To explain why values matter, let us consider the example in Figure 2. Even though the SWMR invariant holds, if during the first read-only epoch Cores 2 and 5 can read different values, then the system is not coherent. Similarly, the system is incoherent if Core 1 fails to read the last value written by Core 3 during its read–write epoch or any of Cores 1, 2, or 3 fail to read the last write performed by Core 1 during its read–write epoch. Thus, the definition of coherence must augment the SWMR invariant with a data value invariant that pertains to how values are propagated from one epoch to the next.

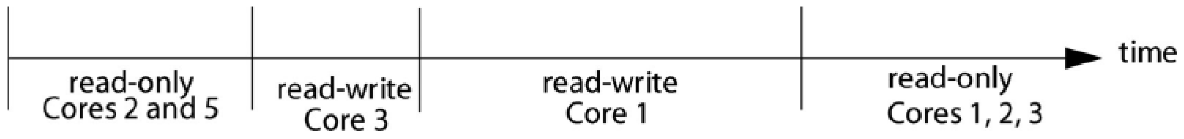

Figure 2: Dividing a given memory location's lifetime into epochs

A coherence protocol's goal is to maintain coherence by enforcing these invariants. To implement these invariants, we associate with each storage structure—each cache and the memory—a finite state machine called a coherence controller. The collection of these coherence controllers constitutes a distributed system in which the controllers exchange messages with each other to ensure that, for each block, the SWMR and data value invariants are maintained at all times. The interactions between these finite state machines are specified by the coherence protocol.

Our cache coherence protocol is a variation of directory protocols. In a directory protocol, the directory maintains the state of each block, and cache controllers send all requests to the directory. The directory either responds to the request or forwards the request to one or more other coherence controllers that then respond. In most directory protocols, a coherence transaction is ordered at the directory. Multiple coherence controllers may send coherence requests to the directory at the same time, and the transaction order is determined by the order in which the requests are serialized at the directory.

We can specify a controller as a table in which rows correspond to block states and columns correspond to events. We refer to a state/event entry in the table as a transition, and a transition for event E pertaining to block B consists of (a) the actions taken when E occurs and (b) the next state of block B. We express transitions in the format "action/next state" and we may omit the "next state" portion if the next state is the current state. Table 1 and 2 show the specification for Cache controller and Directory respectively. Empty entries in tables denote impossible transitions.

States	Events			
	Load or store	Write-back Request	Invalidate	ACK
VALID	Perform load or store	Send data to memory /WAIT_WB	Reply with data and ACK /INVALID	
INVALID	Request permission /WAIT_RDWR		Do Nothing	
WAIT_RDWR	Stall load or store	Stall write-back		Copy data into cache, perform read or write /VALID
WAIT_WB	Stall load or store	Stall write-back	Do Nothing	- /INVALID

Table 1: Cache Controller Specification

States	Events		
	Request (MSG_REQUEST)	Write back (MSG_WB)	ACK
I	Send data to requester, update the owner/ V		
IV	Cannot be processed now, Process the request later	Copy data, send ACK, Push the data onward to whoever is waiting for it / V	Get data and send it to the cache requesting it / V
V	Send invalidation to current owner, put address in waiting state / IV	Copy data, send ACK, make directory the owner / I	

Table 2: Directory Specification

Explanation of protocol:
Initially, all cache blocks and memory blocks are in state *I(dle)/INVALID*. This means that the owner of the memory block is the directory and no core has permission to read or write.

For each of the actions of the core or directory, the protocol is defined as follows:
(For read or write event and memory address state is VALID)
If the memory address the core wishes to read from or write to is marked as VALID, the cache already has permission for that block and can perform the action. If the block is marked as INVALID, the cache controller sends a request (MSG_REQUEST) to the directory for that address and sets the state to WAIT_RDWR.
The directory for that address will get the request from the controller. If no cache owns the requested memory block and no cache is waiting for the data in that memory block (i.e. the state

is I), the directory will send the data to the requesting cache, update the information for the memory block (updating the owner for the address and setting the state to V). Note that the directory processes the request one by one and if it receives a request while another cache is waiting for a particular block, it will not process the request.

If the memory address state in the directory is V, meaning another cache owns the block, the directory will send an invalidation message (MSG_INVALIDATE) to the current owner and postpone giving the read/write permission to the current requester until getting the current data for that block and an invalidation acknowledgement (MSG_ACK) from the current owner. While it is waiting for the acknowledgement, the directory sets the state of the block to IV (i.e. waiting state). During this period, any other request for that block cannot be processed and will be delayed.

(For read or write and memory address state is WAIT_RDWR)
If cache controller is in *IV* mode and gets a load or store, it should delay these requests since one is already in progress.

(For write-back request from core)
If the cores wishes to free space in the cache and put a block back in memory to replace with a block at another address, the cache controller will issue a write-back request (MSG_WB) for the desired memory address to the directory, send the data for the address to the directory and set the state to WAIT_WB.
The directory upon receiving the message with copy the data into memory and send an acknowledgement message to the cache controller. The cache controller will set the state to INVALID upon receiving the acknowledgement message.
If another cache was waiting for the block (state IV), the data is additionally forwarded to whoever is waiting for it. The state is set to V and information about the owner is updated accordingly (i.e. the cache that was waiting). Otherwise, the state of the block is set to I.

(For invalidate events from the directory)
When a cache controller receives and invalidation message from the directory, meaning that another cache controller is requesting the block in memory, the controller will reply with the data for the address and an acknowledgement message. It then sets the address to INVALID.
(Other specifications of the the protocol are shown in the table.)

Remarks:
For this project, you can assume cache controllers sent requests through individual request channels and receive replies through individual reply channels (i.e. each controller has two channels for communicating with the directory, one for sending requests and acknowledgments from caches to directory and one from getting the replies from directory).

The cache controller is supposed to delay the processing of a write-back request or load/store request when it is in a WAIT_* mode until it can process it, but for this project you can ignore the incoming load/store and write-back events.
For simplicity of the project, you can assume the size of all the caches and memory is the same, a single bit, so there is no need for address mapping.

Here is a list of what you should do for this project:

1. Construct Promela model of the described cache coherence protocol by filling the gaps in the code provided (see course webpage). Note that you can change the code if needed but your model should capture the essential details of the protocol as described in the previous Section. Your model should be parameterized by the number of caches so that you can instantiate different configurations easily. One aspect you should think about is: how large should the buffers be for the links between nodes?
2. For some specific configurations, simulate the model and observe its behavior(for example when there are two cores and when there are three cores, try to observe different scenarios like when two cache controller send a request for the same block held by another cache) . Running the simulation repeatedly should change the order in which messages accumulate in various queues and get processed, but hopefully should not violate the protocol invariants.
3. There is a monitor process *monitorSWMR* in the code which can be used to ensure that number of cores having permission to read/write is less than or equal to one at any time. You should use assertion to specify some other safety properties. Run the verifier to check that the model satisfies these specifications for the initial configurations chosen in part (2) above
4. Set up an experiment to check how the number of reachable states of the model, and time and memory used by the verifier grows with the number of caches. For this purpose, change the number of caches from 2 to 6, and plot the above quantities.

You should submit (a) the Promela model of the protocol constructed in part 1, (b) a brief explanation of what configuration and specific scenarios you simulated in part 2, (c) the safety properties and result of verification in part 3, and (d) the tables/charts showing experiments in part 4.

References

[1] Daniel J. Sorin, Mark D. Hill, and David A. Wood, "**A Primer on Memory Consistency and Cache Coherence**." Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, May 2011

Additional Files:
spin-template: sketch of the cache coherence protocol in Promela
TrainController2: Encoding of the controller model of Section 3.1 in Promela