Design and Verification of a Cache Coherency Protocol

Due: Mon. 3/26 11:59pm (Waypoint due via Canvas on 3/12)

Overview

In this assignment, you will design and verify a cache coherency protocol for a multiprocessor system. Your protocol will be a fairly simple invalidation-based protocol, but to get full credit you must implement an optimization. We will describe the basic requirements and a possible optimization for you. As always, creativity is encouraged.

Writing a cache coherency protocol is reasonably challenging; verifying its correctness is a necessary, but very difficult aspect of the process. Sophisticated protocols (e.g., DASH), have been developed and verified, however this remains an active research area. How do we reduce the number of messages for a transaction? The size of the directory? How fast can the directory controller run, and how can we reduce the design complexity? All of these are fundamentally related to the design and verification of the protocol itself.

Baseline Protocol

You will design and verify an invalidation-based cache coherency protocol. The protocol you develop will have a number of characteristics:

1. It uses an interconnect network that supports only point-to-point communication. All communication is done by sending and receiving messages. **The interconnect network may reorder messages arbitrarily**. It may delay messages, but it will always deliver messages eventually. Messages are never lost, corrupted or replicated. Message delivery cannot be assumed to be in the same order as they were sent, even for the same sender and receiver pair.
2. At the receiving side of the interconnect system, messages are delivered to a receive port. Once a message has been delivered to the receive port, it will block all subsequent messages to this port until the message is read. Consider this behavior equivalent to that of a mailbox with room for only one letter: you have to remove the letter from the mailbox before you can receive the next one. On the sending side, there is no such restriction: you can always send messages. The interconnect system has enough buffer space to queue messages.
3. For the purpose of this assignment, you may assume that there is no limit on the buffer space in the interconnect system. However, your protocol will be considered broken if there is a way to generate an infinite number of undelivered messages. Besides, you will not be able to verify your protocol in this case.
4. You may assume that the interconnect network supports multiple lanes. For each lane, you have a separate set of send- and receive-ports for each unit. Traffic on one lane is independent of traffic in the other lanes. Messages will never switch lanes. Note that using fewer lanes is better.

5. Each processor has a dedicated cache that is **not shared with any other processor**. All caches must be kept coherent by your cache coherency protocol. Processors may issue load and store operations only. Because this assignment only deals with cache coherency and not with consistency issues, **you will be concerned with only one storage location** (address). However, you need to model cache conflicts. To do this, you need to model a third operation besides load and store: a cache write-back. Write-backs normally arise from a cache conflict if the old line is dirty. Write-back operations may occur at any time between any pair of load/store operations. If the cache is in a clean state, you may simply set it to be invalid or take the appropriate action according to your CC protocol. Cache replacements of dirty lines must obviously write the line back to memory.
6. You should assume that the coherency unit is equal to one word and that all loads and stores read or write the entire word.
7. Besides processors with their caches, there is one memory unit in your system. The memory unit has a directory-based cache-consistency controller which ensures that only one processor can write to the memory block at a time (exclusive-ownership style protocol). The directory representation is unimportant for this assignment. You should assume that you have a full directory (bit vector) that can keep track of all sharers.
8. The interconnect system can send messages from any unit to any other unit. It is OK if your protocol requires that a cache controller has to send a message to another cache controller.

For this assignment, your cache coherency protocol should not worry about consistency issues. Because of that, you may assume that the memory of this machine has only one word. Your protocol has to make sure that loads from up to three (3) processors always return the value of the most recent stores. In this context, this means that loads and stores issued by one processor are seen by that processor in program order.

You are supposed to write a plain, directory-based cache-coherency base-line protocol without any optimizations other than forwarding of invalidations and exclusive ownership. In other words, your baseline protocol should use no more than 3 hops for any transactions. For example assume that processor 3 has exclusive ownership and processor 1 issues a load, then P3 is supposed to send the cache line to the memory and to P1 (forwarding). This is to minimize latency.

The base-line protocol shall deliver data always in the state needed by the requesting processor. In other words do not bother with speculating on supplying data in E-state for a normal load. Thus E-state is always a consequence of a store. Therefore in this case you only have 3 cache states: I = invalid, S = shared (read-only) and M = modified (exclusive and dirty). The memory unit could be regarded as a home node without a processor, so it will never do anything on its own. For example, it will never issue an unsolicited recall-request.

Optimizations

The design space for cache coherency protocols is very large. In the past, a variety of optimizations have been proposed and implemented that reduce the directory storage, cut the number of message hops, or otherwise improve resource and performance for distributed shared-memory systems. In this project, we want you to implement and verify at least one optimization for your baseline protocol.

For example, consider the scenario depicted in Figure 1. Initially, the address X is shared among a group of nodes (not shown). Then, node 1 requests modified (M) access to X. As part of the invalidation they receive, the sharers of X record that node 1 now has X in the modified state. When one of the former sharers (N) requests access to X again, a speculative request is issued to node 1 for X in hopes that P1 still has X in modified state.

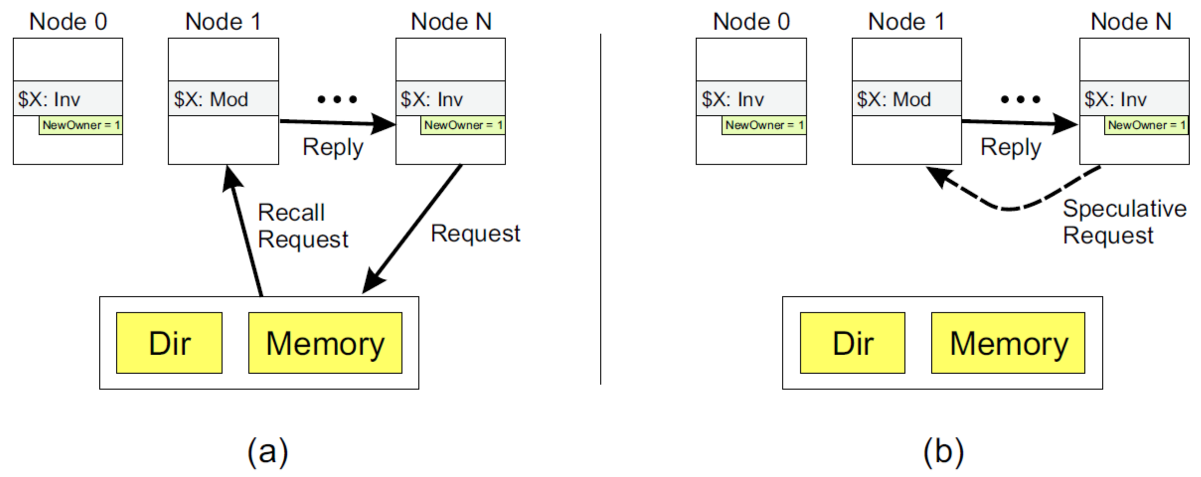

Figure 1. Speculative requests to reduce the number of message hops. (a) shows the baseline protocol with three hops, while (b) reduces this to two hops if the speculative request is successful.

The goal is to reduce the number of hops accessing X. If the speculative request is successful, the normal three-hop transaction is reduced to two. Note that the sharing node N must still send a non-speculative request to the directory controller in case X is no longer held at node 1.

This optimization is fairly straightforward, but note that we're not discussing corner cases here: what happens when the speculative request arrives at the same time that node 1 is writing-back X? You must flush out the details of this or other optimizations and make sure they are correct by verifying the protocol and your optimizations.

Deliverables

You must specify and verify your protocol and optimization(s) using Murphi, a formal verification tool available from the class website page. Murphi runs out-of-the-box on an x86 Linux machine when compiled for 32-bit binary, but contact us ASAP if you have any trouble. The Murphi distribution comes with a manual and tutorial in the doc/directory; there are also some documents under the ex/dash directory worth examining. Using the system, your will turn in the Murphi source code that describes and verifies your protocol and a **two-page description** of your protocol, optimizations, and verification approach.

Deadlines

To prevent you from leaving this to the last minute, we want to see the start of a working protocol by midnight on March 12th. Submit to canvas a short (one-page) note describing your status and what optimization(s) you are implementing. You should have started working with Murphi and made significant progress towards the baseline protocol. A small number of points (10) of your assignment grade will come from completing this waypoint.

The final version is due two weeks later and should be submitted through Canvas (**please zip the files**). We expect you to submit:

(1) A revised two-page description of your protocol, optimizations and verification approach
(2) A diagram documenting the complete state machine for your protocol
(3) The Murphi code that demonstrates the correctness of your protocol. The assertions within this code must prove that coherence has been maintained.
(4) The output from Murphi, showing that no errors were found, the number of states explored and the running time

DO NOT LEAVE THE BULK OF ASSIGNMENT FOR THE LAST FEW DAYS! DO NOT OMIT THE WRITE-UP.