6.823
Computer System Architecture
Lab 3

Assigned Apr. 11, 2020 ***Due Apr. 24, 2020***

http://csg.csail.mit.edu/6.823/

Summary

Many modern multi-core computer systems support shared memory in hardware. In a shared memory system, each of the processor cores may read and write to a single shared address space. Cache coherence protocols, which manage the read and write permissions of data in various caches, are an important component in ensuring the correct operation of shared caches in a multi-core system.

Cache coherence protocols are challenging to design, in particular due to the multitude of races and corner cases. Verifying their correctness is a necessary, but very difficult aspect of the process. Sophisticated protocols have been developed and verified. However, this remains an active research area. In this assignment, we will design and verify a cache coherence protocol for a multi-core system.

To verify our coherence protocol, we will use Murphi, a formal verification tool. Murphi employs model checking to verify the correctness of the specified coherence protocol. Model checking is an automated technique that, given a finite-state model of a system and a set of desired properties, checks if the properties hold for all reachable states of the system, in that model. More concretely, we will describe the finite-state machine corresponding to the coherence protocol in the Murphi description language, and enumerate a set of desired properties or invariants. The Murphi verifier will systematically enumerate the entire space of reachable states, and check that the specified invariants are not violated.

Murphi may be downloaded from the following web page:
http://formalverification.cs.utah.edu/Murphi/ However, we have already provided you with the necessary materials. As always, this lab is to be completed individually. You are encouraged to discuss lab concepts with fellow classmates.

Setting up

To obtain the materials for lab 3, use the following commands, assuming that you start in your individual repository (`cd $USER`) from the previous labs:

```
% cp -r $LAB3FILES ./
% git add lab3handout
% git commit -m "Lab 3 Initial Check-in"
```

In the `lab3handout` directory that was just created, you should find the Murphi source code and a protocol sub-folder, which contains some examples and the framework code to get you started. Type the following at the command prompt:

```
% cd lab3handout
```

First, build the Murphi compiler.

```
% cd Murphi/src
% make mu
% make install
% ln -s ../bin/mu.x86_64 ../bin/mu
```

Let us test a simple model. Run the following commands (assuming you start from `lab3handout`):

```
% cd protocol
% ../Murphi/bin/mu pingpong.m
```

`pingpong.m` contains a simple example written in the Murphi description language. Running it through the Murphi compiler should generate `pingpong.C` file. You can then compile the generated C code:

```
% make pingpong
```

This builds the final verifier, which you can run as:

```
% ./pingpong -v
```

You can find other examples in the Murphi directory (`../Murphi/ex`). The Murphi user manual is a useful resource to understand the language constructs. It is available within the Murphi directory (`../Murphi/doc/User.Manual`).

Lab Task

You will design and verify an invalidation-based cache coherence 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 and delay messages, but it will always deliver messages eventually. Messages are never lost, corrupted or replicated. Message delivery cannot generally 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 mail-box 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.

3. For the purpose of this assignment, assume the interconnect system has enough buffer space to queue messages. If your network runs out of space, you may increase the constant `NetMax` to any finite value. However, this increases the size of the state space, making verification slower. Therefore, if you increase `NetMax`, you should increase it only to the minimum finite value needed for your solution to work, and you should explain this in your answer to the lab questions.
4. You may assume that the interconnect network supports multiple lanes (a.k.a. virtual channels or VCs). For each lane, you have a separate set of send- and receive-ports for each unit. Messages will never switch lanes. Note that using fewer lanes is better.
5. Note that in `msi.m`, VCs have different priorities which affect how messages are received. A node will only receive a message if there are no messages with higher priority. You may try to take advantage of this, but note that a solution exists in which all messages have the same priority.
6. Each processor has a private cache. All caches must be kept coherent by your cache coherence protocol. Processors may issue load and store operations. Because this assignment only deals with cache coherence 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, the processor can initiate a third operation besides load and store: a cache eviction. Evictions 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 coherence protocol. If the cache is dirty (modified), you must write the evicted cache line back to memory.
7. You should assume that the coherence unit is equal to one word and that all loads and stores read or write the entire word.
8. 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 can assume that you have a full directory (bit vector) that can keep track of all sharers.
9. The interconnect system can send messages from any unit to any other unit. It is OK if your protocol requires that a cache sends a message to another cache.

For this assignment, your cache coherence 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 must ensure 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.

The baseline 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 an exclusive state for a normal load. Exclusivity 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.

3-hop vs 4-hop protocol:

A simple incarnation of the MSI protocol is the 4-hop protocol, where the directory is responsible for satisfying all data requests from the processors. Here, all requests for data are satisfied with at most 4 hops (one such 4-hop transaction is show in Figure 1).

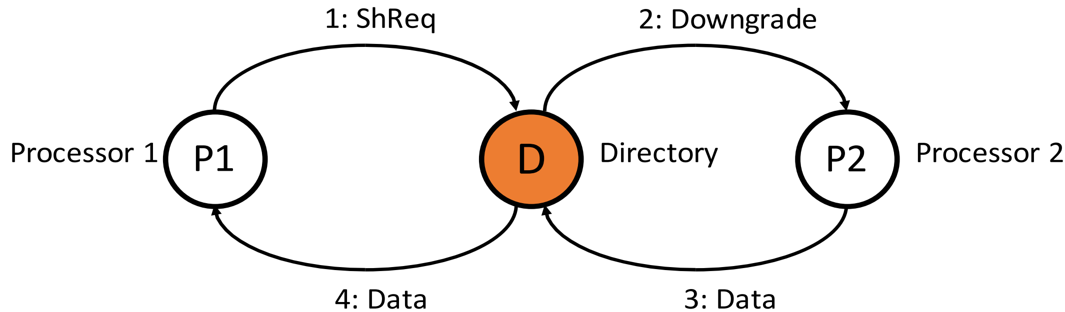

Figure 1: 4-hop transaction. Assume P2 originally has the address in M state. When P1 issues a ShReq, the directory D issues a downgrade request to P2, which writes back the data to the directory, which then forwards it to the requestor P1.

An optimization to reduce the latency of requests, is to allow P2 to respond to P1 directly with the data in the above scenario. The resulting transaction is shown in Figure 2. Allowing such forwarding of requests transforms the protocol to a 3-hop protocol.

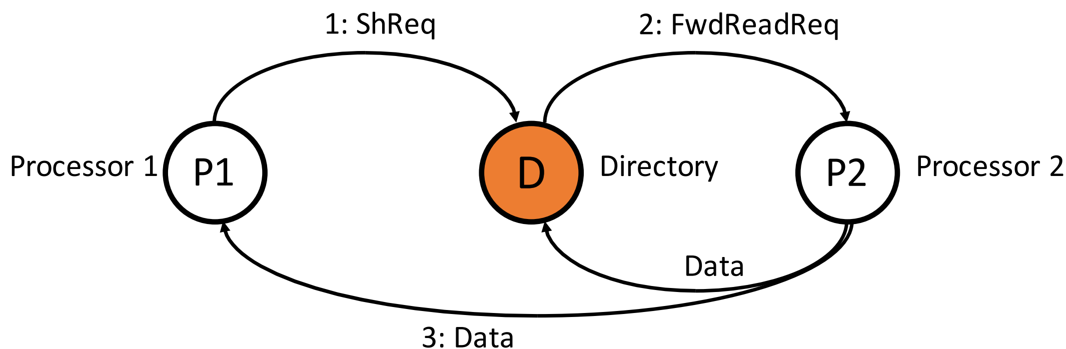

Figure 2: 3-hop transaction. P2 originally has the address in M state. When P1 issues a ShReq, the directory instead of issuing just a downgrade request, sends a FwdReadReq which contains the id of the requestor. P2 then sends the data to both the directory and P1, and also downgrades to the S state.

Your task is to write a 3-hop directory-based cache-coherence protocol based on the MSI protocol discussed in the lecture.

To help you get started we have provided you with framework code in `protocol/msi.m`. In addition, we have also provided a 4-hop Valid-Invalid (VI) protocol in `protocol/twostate.m`. along with some guidelines on how you may proceed in developing your MSI protocol. Since the 3-hop MSI protocol is slightly more complex, you may benefit from starting with the 4-hop MSI protocol, and then enable the forwarding optimization. We will grant partial credit for turning in a correct 4-hop MSI protocol.

Although your solution will not be graded on its performance in terms of wall clock time, you should note that your Teaching Assistant is impatient. It is possible to design a protocol that can be verified within a few seconds. For grading purposes, we will allow your tool to run for up to 30 minutes. After 30 minutes, we will kill your submission and assign a grade accordingly.

When you have completed the lab to your satisfaction, submit your changes to the git repository. The deadline for submission is 23:59:59 EDT 24 April 2020. We'll grade whatever code you have committed and pushed by the deadline. **No Late Submissions will be accepted!** Seriously.

Tips and Hints

Protocol design:

You might want to use pencil and paper and draw out timelines. Identify stable state(s) you'd like to reach, and work to ensure those states are reached in a few steps if no more requests occur.

Sometimes races need to be solved by making a particular agent responsible for recognizing the race, and deciding which actions get prioritized. Sometimes you need to make an agent wait for acknowledgements before they are allowed to make other requests. If long-delayed messages are received after they are no longer relevant, consider whether you could have made some agent wait for that message earlier, or perhaps made some agent wait for an acknowledgement of that message. Try to be consistent in your choices of how actions are prioritized.

In some states, you may want to refuse to process certain requests until some pending action completes. You can set `msg_processed := false` to make the network hold a message for you to process later. You can think of this as being equivalent to responding a message by sending a NACK, and having the recipient of the NACK retry sending the original message at a later time. You do not need to explicitly model NACKs or retry messages.

Debugging:

Throughout the course of the lab, you will often have to debug your protocol design by looking at which sequence of actions resulted in violation of an invariant. Murphi does not

output the offending trace by default, so you should run your protocol with the following option if you want to view the error trace:

```
% ./msi -tv
```

The `-tv` option produces the shortest trace that violates any invariant. Note that there are also other command line options that you may find useful in the Murphi user manual (see Section 2.5).

When examining a trace, you may find that things start to go wrong but a violation of an invariant does not occur until several steps later, resulting in a long and complex trace. To make debugging easier, you may add invariants beyond those we have provided. However, do not change the provided invariants.

During debugging, you may wish to temporarily decrease `ProcCount` from 3 to 2 and see what happens in the simpler case of two processors.

Drawing graphs:

You will have to draw state transition diagram(s) to describe your protocol in the lab report (see next section). You may find it convenient to use PowerPoint to draw the diagrams and convert them to images/pdfs. If, however, you want to use visualization tools geared towards graphs, here are a couple:

- Graphviz (http://www.graphviz.org/) is a popular tool visualizing graphs, especially directed graphs. You just specify the nodes, edges, labels, etc, and it will automatically try to place the nodes in a sensible way and then draw the edges as splines that curve around so that edges don't cross over nodes or labels. This tends to result in figures that look spaghetti-like at a glance and are distinctively automatically generated, but are actually decent for automation purposes: http://stackoverflow.com/questions/6714068/tighten-the-dot-graph-making-it-more-symmetric
- If you're writing your report in Latex, several packages exist that are capable of drawing diagrams and schematics. See: https://tex.stackexchange.com/questions/20784/which-package-can-be-used-to-draw-automata

Lab Deliverables

Your final deliverable will be a 3-hop MSI protocol, specified and verified using the Murphi language. You should also turn in the following:

(1) A description of your protocol.
(2) State transition diagram(s) documenting the complete state machine for your protocol.
(3) The output from `./msi -v` showing that no errors were found, the number of states explored and running time.

When you have answered these questions to your satisfaction, put them in a file called lab3questions.pdf (or lab3questions.doc) in your lab3handout directory, then run the following to add them and commit them.

```
% git add lab3questions.pdf
% git commit -m "Lab 3 Questions Check-In"
% git push origin master
```

As with the lab code, we'll grade whatever you have checked in by the deadline.

Lab Grading

10%: Submission compiles
60%: Grade based on your protocol
  40%: Correct 4-hop MSI protocol
  60%: Correct 3-hop MSI protocol
30%: Quality of lab response

Advice on Mine Sweeping

There may be bugs in either our code or infrastructure. If you notice any `interesting' or `unexpected' behavior it could be a problem in the code or infrastructure that we provided. Report these bugs immediately to the TAs. This will help to ensure prompt fixing of any issues that may arise.

Guides for the perplexed

http://formalverification.cs.utah.edu/Murphi/ - Murphi home page
https://help.github.com/articles/git-and-github-learning-resources - Git learning resources
https://git-scm.com/book/en/v2 - ProGit ebook

Acknowledgements

Brian T. Gold, Carnegie Mellon University