Assignment 2: Design Verification of a Cache Coherence Protocol

1 Introduction

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

Writing a cache coherence protocol is reasonably challenging; verifying its correctness is a necessary but very difficult aspect of the process. Sophisticated protocols (e.g., Dash [5]), have been developed and verified, and this remains an active research area [1, 4, 6, 7, 8, 9, 10]. 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.

2 Baseline Protocol

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 messages arbitrarily. It may delay messages, but it will always deliver messages eventually. Messages are never lost, corrupted or replicated. Messages 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 to 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 system supports multiple lanes. For each lane, you have a separate set of send and receive ports for each unit. Traffic on one lane in 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 coherence protocol. Processors may issue load and store operations only. Since 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, 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 cache coherence protocol. Cache replacements of dirty lines must obviously write the line back to memory.

6. You should assume that the coherence unit is equal to one word and that all loads and stores read and 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 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 has to make sure that loads from up to three 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 coherence baseline protocol without any optimizations other than forwarding of invalidation and exclusive ownership. In other words, your baseline protocol should use no more than three hops for any transaction. For example, assume that *processor-3* has exclusive ownership and *processor-1* issues a load, then *processor-3* is supposed to send the cache line to the memory and to *processor-1* (forwarding). This is to minimize latency.

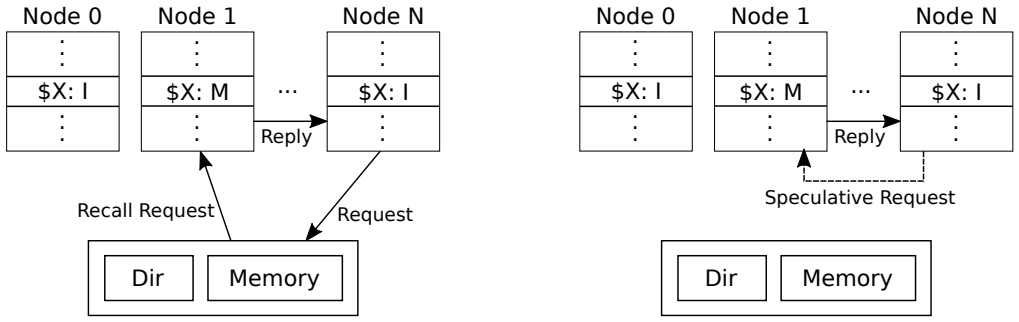
The baseline protocol shall deliver data always in the state needed by the requesting processor. In other words, do not bother with speculating on supplying data in E state for normal load. Thus, E state is always a consequence of a store. Therefore, in this case you only have three cache states; I for invalid, S for shared (read only) and M for 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 Optimization

The design space for cache coherence 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 assignment, 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 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 N sharers requests access to X again, a speculative request is issued to *node-1* for X in hopes that *processor-1* still has X in modified state

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 if X is no longer held at *node-1*.

This optimization is fairly straightforward, but note that we are 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.



(a) Without optimization (b) With optimization

Figure 1: Coherence protocol optimization example

4 Infrastructure

You can get the tarball for the assignment from `/cad2/ece1755s/assignment2.tar.gz`. In it you will find the source files for the $Mur\phi$ [2, 3] verification tool, its documentation as well as some examples.

$Mur\phi$ is a compiler that takes the coherence protocol state machine written in the $Mur\phi$ language as an input. The compilation step results in a `.C` file that contains the source code for the verifier to be run. After that you should compile the resulting `.C` normally with a `c++` compiler and run the resulting executable file to run the verifier. (For more details check section 2.2. in `doc/User.Manual`).

You are highly encouraged to go through the documentation but just to get you started here are the basic steps to use $Mur\phi$:

1. ~~Build the compiler~~[1]: You can skip this step and use the binary provided in `bin/mu.x86_64_prebuilt`. However, if this does not work and you need to recompile the source code then go to `src/` and run `make` then `make install`. This will build the compiler and install it into `bin/mu.x86_64`.

2. Compile your design: You have been provided with two examples in the `ece1755_sample` folder. From this folder, run `../bin/mu.x86_64 twostate.m` for example. This produces the verifier source code, `twostate.C`.

3. Build the verifier executable: Compile the resulting `twostate.C` with a normal `c++` compiler. To make it easier, you can use the provided `Makefile`. So, run `make twostate` and you should get the executable in the same folder.

4. Finally, run the resulting `twostate` executable and start analyzing the results.

5 Deliverables

You are to complete this assignment individually. You must specify and verify your protocol and optimization(s) using $Mur\phi$, a formal verification tool. $Mur\phi$ 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 $Mur\phi$ 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, you will turn in the $Mur\phi$ source code that describes and verifies your protocol and a two-page description of your protocol, optimizations and verification approach.

To encourage you not to leave this assignment to the last minute, you are required to submit a one-page report describing your status and what optimization(s) you plan to implement. To submit this milestone

[1] Due to updates to the `ug` machines, compiling then running $Mur\phi$ on the `ug` machines results in a segmentation fault. Therefore, we recommend using the pre-built binary directly.

report use the following:
`submitece1755s 2 milestone.pdf`

The final submission should include the following:

1. `report.pdf`: A three-page report including:
   (a) A description of your protocol, optimizations and verification approach
   (b) A diagram documenting the complete state machine of your proposal
2. `protocol.m`: The $Mur\phi$ code that demonstrates the correctness of your protocol. The assertions within this code must prove that coherence has been maintained
3. `output.txt`: The output from $Mur\phi$ showing that no errors were found, the number of states explored and the running time
4. `milestone.pdf`: A document describing progress by the time of the milestone

The command to submit your work should be similar to the following:
`submitece1755s 2 protocol.m report.pdf output.txt`

Please adhere to the provided naming conventions Misnamed files will not be marked. You can view the files you submitted using the command `submitece1755s -l 2`. Finally, while developing, remember not to leave the Unix permissions to your code open.

6 Due Date and Marking Scheme

We strongly encourage you to have started working with $Mur\phi$ and made significant progress toward the baseline protocol by February 17, 2023. The deadline for the complete submission is 11:59 PM March 3, 2023.

The grade breakdown is as follows:

- 20% - final report
- 55% - baseline protocol $Mur\phi$ implementation
- 20% - optimized protocol $Mur\phi$ implementation
- 5% - output file

7 Guide

- `http://formalverification.cs.utah.edu/Murphi`

Acknowledgement

Brian T. Gold, Carnegie Mellon University

References

[1] J. Alsop, W. T. Na, M. D. Sinclair, S. Grayson, and S. Adve, "A case for fine-grain coherence specialization in heterogeneous systems," *ACM TACO*, vol. 19, no. 3, 2022.

[2] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *Proc. ICCD*, 1992.

[3] D. L. Dill, "The murphi verification system," in *Proc. CAV*, 1996.

[4] A. Franques, A. Kokolis, S. Abadal, V. Fernando, S. Misailovic, and J. Torrellas, "WiDir: A wireless-enabled directory cache coherence protocol," in *Proc. HPCA*, 2021.

[5] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proc. ISCA*, 1990.

[6] N. Oswald, V. Nagarajan, and D. J. Sorin, "ProtoGen: Automatically generating directory cache coherence protocols from atomic specifications," in *Proc. ISCA*, 2018.

[7] N. Oswald, V. Nagarajan, and D. J. Sorin, "HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols," in *Proc. ISCA*, 2020.

[8] N. Oswald, V. Nagarajan, D. J. Sorin, V. Gavrielatos, T. Olausson, and R. Carr, "HeteroGen: Automatic synthesis of heterogeneous cache coherence protocols," in *Proc. HPCA*, 2022.

[9] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *Proc. HPCA*, 2012.

[10] G. Zhang, W. Horn, and D. Sanchez, "Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems," in *Proc. MICRO-48*, 2015.