CSE471 Homework Assignment #2: Cache Coherence

Due Tuesday, May 17th, before lecture

Milestone 2: Implementing the 4-State Protocol

The second milestone requires you to implement your new protocol in a simulator, and to ensure it works by running it on a set of benchmarks.

Getting Started with the Simulation Infrastructure

In this assigment, we will be using a Pin-based simulator called MultiCacheSim. You will be writing a cache coherence plugin for MultiCacheSim that implements your 4-state protocol. There is some shared infrastructure that we will be using, so you will need to work on attu, or any of the department machines that have access to the shared filesystem.

First, download the code archive from the course website. To get you started on your implementation, we have provided you with a plugin that implements the 3-state protocol we studied in class (MSI_SMPCache). It will benefit you to read this code carefully to understand how the simulator works. Brandon will discuss the 3-state protocol's implementation during Section.

To build the 3-state protocol plugin, run `make MSI_SMPCache.so`.

To run MultiCacheSim and load this cache coherence plugin, run:

```
/cse/courses/cse471/11sp/pin -t
/cse/courses/cse471/11sp/HW2-CacheCoherence/Release/MultiCacheSim_PinDriver.so
-protos ./MSI_SMPCache.so -numcaches 8 -- /usr/bin/gcc
```

This command will run `/usr/bin/gcc` in a simulation of 8 processors with caches that are kept coherent by the protocol implemented in MSI_SMPCache.so. You can vary the number of caches that are simulated by changing the *numcaches* command line argument. You can specify a number of protocol plugins as a command line argument to the *-protos* option as a comma separated list.

Protocol Plugins

A protocol plugin keeps caches in a MultiCacheSim simulation coherent. The provided MSI_SMPCache is an example that illustrates the essential components of a protocol plugin.

A protocol plugin must implement 4 interface methods:

1. readLine(readPC, addr)
2. writeLine(writePC, addr)
3. readRemoteAction(addr)
4. writeRemoteAction(addr)

`readLine()` and `writeLine()`
The readLine and writeLine methods implement protocol actions taken in the event of a read or write. These methods must do several things:

1. Check the tags of the block to determine if the access is a miss
2. Call `readRemoteAction()`, which implements the snoop in remote caches
3. Determine the correct state in which to cache the block being accessed
4. Cache the block

`readRemoteAction()` and `writeRemoteAction()`
These two methods implement snooping of remote transactions on the bus. The methods are called by a simulated processor when it makes a memory access. In these methods the processor iterates through the other caches, updating their state as though they have snooped its memory access. `writeRemoteAction` is called from within `writeLine()` and `readRemoteAction` is called from within `readLine()`.

These methods return messenger objects that provide information about coherence state. `readRemoteAction` returns an object with two fields: `isShared` and `providedData`. `isShared` should be true if the line being accessed is in *shared* state in a remote processor. `providedData` should be true if the snooping processor put its data on the bus for the accessing processor. The fields of this messenger object can be used to determine whether an access was serviced by a remote cache, or by memory.

`writeRemoteAction` returns an object with one field, `empty`, which is not necessary for your simulations.

Implementing Your Protocol Plugin When building your protocol plugin, you should work from the provided protocol plugin as a base. It implements most of the functionality you will need, and illustrates many helpful simulator mechanisms (how to get a cache line's state, for example). You should be able to implement your protocol by understanding the implementation of the four methods that we've given you, and changing the protocol logic to implement the fourth state.

We will provide you with a reference solution. You can run your plugin alongside the reference solution plugin, and the two implementations should emit the same final output. The reference solution will give you a way to check your work, and be sure that the protocol you end up implementing is correct.

Benchmarks For this milestone, you will familiarize yourself with the benchmarks in the PARSEC multithreaded benchmark suite, and then use them to both test your implementation and gather data for the quantitative analysis that will be part of the third Milestone. PARSEC is great for debugging, because it comes with a test harness program that allows you to easily execute its programs within Pin-based simulations. The command for doing so is:

```
/cse/courses/cse471/11sp/parsec-2.1/bin/parsecmgmt
-a run -p <benchmarkname>
-d <workdirectory>
-n 8 -s '/cse/courses/cse471/11sp/pin
-t /cse/courses/cse471/11sp/HW2-CacheCoherence/Release/MultiCacheSim_PinDriver.so
-protos ./MSI_SMPCache.so,./FOO_Protocol.so -numcaches 8 -- '
```

where <workdirectory> is a writable working directory, and <benchmarkname> is one of:

- blackscholes
- bodytrack
- freqmine
- swaptions

- fluidanimate
- vips
- x264
- canneal
- streamcluster

The `parsecmgmt` command is self-documenting if you run it with no options. Note the single quotes used in the `-s` option.

Your quantitative analysis that is part of the third Milestone (not this one) should compare the performance and the scalability of the 3-state and 4-state protocols. You'll use the output provided by the simulator to do this analysis. Be sure to utilize the component metrics, so that you know not just which protocol works best, but *why*. Although the actual analysis is part of the report you'll turn in for the next Milestone, you might want to get a head-start on the simulations you'll need.

What to Turn In

You are responsible for turning in two things:

- The code to your protocol plugin (header and implementation)
- A document containing a table of the data produced by your protocol simulation after running all the benchmarks. Each cache's output is comma-separated value format, so you should be able to easily tabularize it. Keep in mind that Brandon will still be running your code along side the reference, in spite of the fact that you're turning in your output.