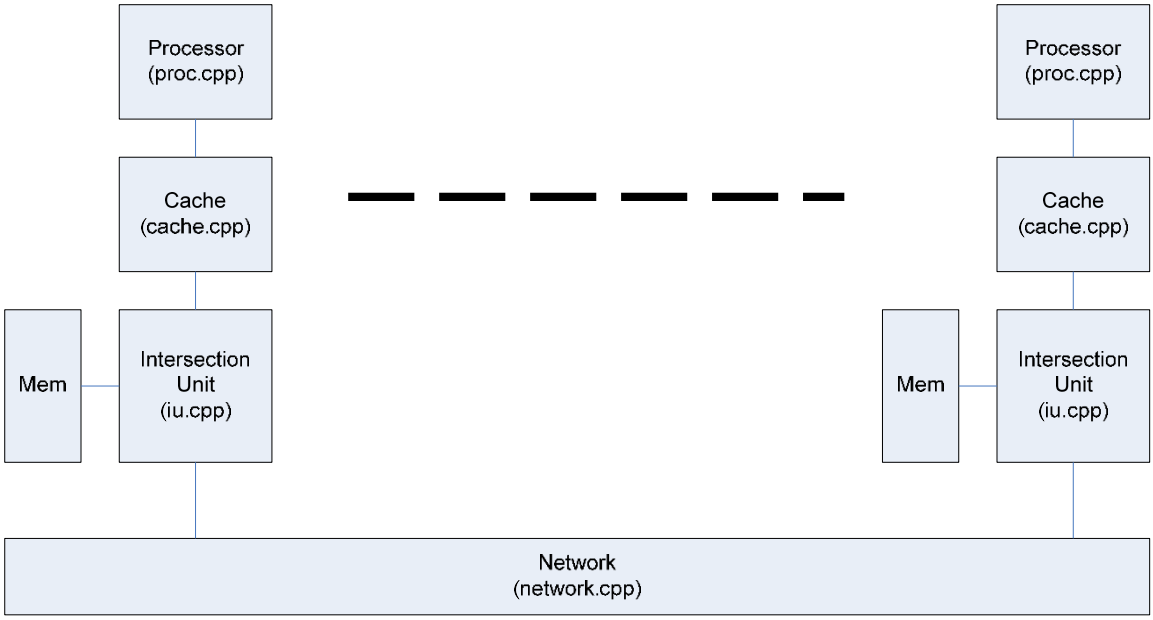
ECE382N-10: Parallel Computer Architecture
2023 Fall
Derek Chiou, Instructor
Mengtian Yang

Lab 2: Cache Coherency

READ ALL OF THIS DOCUMENT AND THE CODE BEFORE STARTING

Due October 30, 2023 at 2:59PM Central Time

Your task is to implement an Interface Unit (IU) that supports the MESI protocol. Below is a picture of the machine you are simulating and the name of the file of the code that describes that particular component.



The lab code can be found on Canvas. It compiles and runs on both Ubuntu Linux (and, I suspect, most other Linuxes) as well as Cygwin (www.cygwin.com), provides Unix-like environment on top of Windows.) On Cygwin, the executable will automatically be sim.exe. The code is simply a skeleton with which will form the basis of your code. It currently implements incoherent cached shared memory with no writeback.

You'll see a FYTD (For You To Do) comment sprinkled around the code. This is where I expect that you need to change the code. You may need to change other places as well, and you might not need to change code at each FYTD, depending on how you tackle the problem.

Some simplifying assumptions:

1. Rather than try to model everything accurately timing-wise, timing is only modeled in the network.
2. The IU is only capable of receiving at most one network message per cycle and sending one network message per cycle. The cache can only send one request to the IU (using the from_proc() in IU) per cycle and the IU can send at most one operation (snoop or reply) to the cache per cycle. Only proc can do cache->load/store, only IU can do cache->snoop/reply. You cannot directly load from/store into the cache from the IU.
3. The IU can access memory (including the directory) no more than 6 times per cycle, where each access is no more than one cache-line large. If you add a queue or two, I won't count accesses against them towards the 6 memory accesses, but be reasonable. These constraints are to help model the constraints of a real system. Your grade will be impacted if you don't follow these constraints.
4. You may need additional space to implement your directory information, queues, etc. You can either implement additional structures within iu.h, or implement the directory within the last ¼ of the mem. The additional space you use should be no larger than ¼ the space of the memory, regardless of whether you use a separate directory structure or implement it in mem. Therefore, addresses used for loads and stores during testing will be below 24K (0-24K, the machine is word addressable). If you use the mem structure for your directory, the addresses for the directory should be 24K and above (24K-32K).
5. Cache lines are each eight 32bit words. Local memory is 1024 cachelines, each cacheline being 32B, for a total of 32KB. The global memory address space is the number of processors * local memory.
6. Excluding communication with other modules, all processing within the processor, cache, and intersection unit (IU) happens in a single cycle.
7. Only one outstanding cache miss per processor can be in progress at any given time (a fully blocking cache.)
8. There is only one core/processor per network node. This means you don't need to maintain snoopy coherence within a single node.
9. You need to support up to 32 processors. You may assume the number of processors is a power of 2.
10. You must use gen_node to generate the node number from the address.
11. The cache is fully blocking. That means if there is a cache miss, the cache blocks, not allowing the next load/store to be issued until the cache miss is resolved. Thus, loads and stores will be in order and only issued one at a time per processor.
12. As iu_t::advance_one_cycle() starter code shows, the IU can only handle ONE request, from any interface,each cycle.
13. If you are broadcasting / multicasting invalidates, you can push multiple requests into the network in a cycle. However, you still can only pop one network or one cache request in the IU per cycle. So the packet rate to the network does not change.

Your code should be well documented and as elegant as possible. Use as much of the infrastructure provided as possible. We would recommend spending some time reading my code as the simulator is written in a particular style that you might not be used to. **You are only allowed to modify cache_snoop.cpp, iu.* and test.*. We will replace all other files to the starter version during test.**

Part 1: Warm Up (5%)

Compile and run the code with the following parameters:

```
./sim 1 10000 0 0
```

Why isn't the hit rate 0.5 or so? What would be a reasonable fix to get the hit rate, in this case, to 0.5? You don't need to actually implement the fix.

Part 2: RWITM (5%)

Note that the bus request created when you store to a location that is Invalid in the cache is exactly the same as the bus request created when you store to a location that is Shared in the cache. This is done for simplicity's sake (and I would strongly suggest that you stick with it.) Why is this approach simpler? How is it normally done and why is it normally done that way?

Part 3: Add Writeback (10%)

The current code does not writeback modified cache-lines that are being replaced. Add support to writeback the cache-line. Remember the IU restrictions when you do so.

Part 4: Add Snoop (10%)

Implement snoop in cache_snoop.cpp.

Part 4: Directory Protocol (35%)

Design and implement a directory-based scheme. Maximize performance while minimizing directory state.

Part 5: Test code (35%)

Verification is very important, thus accounting for the large fraction of the grade. Write as many test cases as you feel is necessary to thoroughly test your code. Isolate your test to the test.cpp and test.h files (We plan to use your tests on others' code. You will receive more points here if you catch others' bugs.) Thus, you are to write architectural level tests for tests that will be used to test other student solutions (e.g., don't write tests where you expect a specific response at cycle 37, or expect to be able to look into internal state.) Your submitted tests should only do loads and stores. Think of it like you are writing the software that runs on the cores, so you can't see the internal state of the hardware. This is a way to ensure your code works and to help us check your classmates. Tests should either call ERROR/ERROR_ARGS, or print out "passed". Document your test cases.

How we will test

We plan to use your tests on your fellow students' code and use their tests on your code. Testing will be done by copying our (or your classmates') test.*, and your cache_snoop.cpp, iu.*, to our starter code , making, and then running the test(s).

How to submit

Please submit Lab2 through Canvas. Be sure to indicate your names as well as who did what if there was any division of labor. Please work together as much as possible as you will all be responsible for all of the material.

Hand in your Lab2 in the correct format. It should be a tarball of all of the files of your problem set. The tarball should be labeled Lab2_LastName1_LastName2.tgz. The last names should be in alphabetical order. The tarball should contain a single top-level folder named Lab2_LastName1_LastName2 and follow the exact structure of the lab code provided. Only ONE member of the group should submit.

Your documentation should live inside of the doc/ directory and be called Lab2_LastName1_LastName2.pdf. Include your documentation source as well. Your code should live in the sim/ directory.