CS {4/6}290 & ECE {4/6}100 - Spring 2024
Project 3: Cache Coherence

Dr. Thomas Conte and Pulkit Gupta

Due: April 23rd @ 11:55 PM

Version: 1.0

Changelog

1. Version 1.0.0 (2024-04-13): Initial release

1 Rules

- **This is an individual assignment. ABSOLUTELY NO COLLABORATION IS PERMITTED.** All cases of honor code violations will be reported to the Dean of Students. See Appendix A for more details.
- The due date at the top of the assignment is final. Late assignments will not be accepted for full credit. Please see the syllabus for the late policy for this course.
- Please use office hours for getting your questions answered. If you are unable to make it to office hours, please email the Head TA/Instructors.
- This is a tough assignment that requires a good understanding of concepts before you can start writing code. **Make sure to start early.**
- Read the entire document before starting. Critical pieces of information and hints might be provided along the way.
- Unfortunately, experience has shown that there is a high chance that errors in the project description will be found and corrected after its release. **It is your responsibility to check for updates on Canvas, and download updates if any.**
- Make sure that all your code is written according to **C11 or C++20** standards, using only the standard libraries.

2 Introduction

This project will have you implement the agent and directory controllers for some cache coherence protocols on a multiprocessor. The underlying notion is straightforward: all caches must see all writes on a piece of data in the same order. The implementation of this idea is obviously not so simple (otherwise we probably wouldn't have a project on it!). Your modifications to the simulator framework will maintain coherent caches for the **MSI, MESI, MOSI, and MOESIF** protocols using a directory-based approach. You are provided the agents for MSI and MOSI, and the directory controllers for MESI, MOSI, and MOESIF. **You will need to implement the agents for MESI and MOESIF, and the directory controllers for MSI**. Your simulator will simulate systems with 4, 8, or 16 processors (cores) and will need to ensure that coherence is maintained. You have a completed/working pair of agent and directory for MOSI. 50% of each of the remaining protocols has been implemented and as such this project is not expected to be as difficult or time consuming as the other projects in this course once you understand how directory based coherence works. Detailed comments have been provided in the directory files you need to implement. The agent transitions are available in Appendix C

Note: This project MUST be done individually. Please follow all the rules from Section 1 and review Appendix A.

3 Simulator Specifications

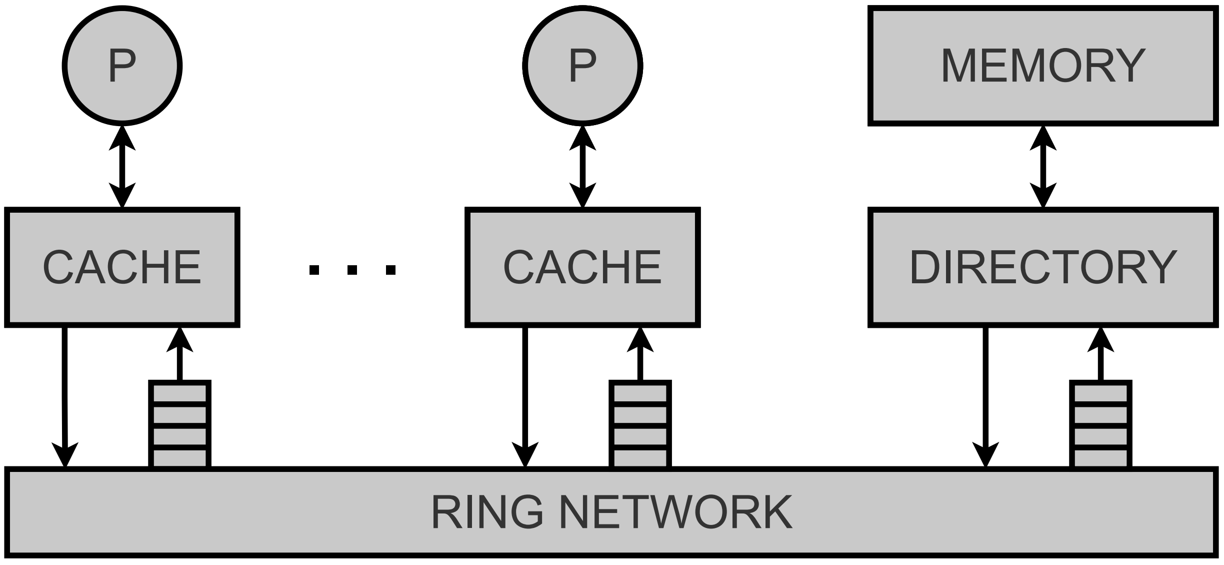

Figure 1: System Overview

3.1 Simulator Overview

This section details some of the specifics of the provided code in `simulator/`. Deeper implementation details are not covered in depth since they should not affect your work in the `coherence/` directory. **Remember that you are not going to be simulating data values at any point in the simulation.**

1. The system consists of 4, 8, or 16 processors (cores). Each core is essentially a memory trace reader that will read in the provided trace files. You will not need to modify these structures.
2. For the sake of simplicity, each core has a single L1 cache that is fully associative, infinite sized, and has a single cycle access latency. The code for this cache is provided. These caches are not coherent as of now, so you will need to implement the agents and directory controllers to make the caches coherent.
3. The system has a single memory controller which accesses the off-chip memory with a 100 cycle latency. The centralized directory is kept at the memory controller and responds to one request/message per cycle. If the data is not stored at any of the caches or if the data needs to be read from memory, the 100 cycle DRAM access penalty must be paid. You will be implementing the directory controller for the MSI coherence protocol.
4. The caches and centralized directory are connected by a deterministic ring network. This means that messages may take a number of cycles to get from their source to their destination. Each entity/node on this network (cache/agent or directory controller) has an unlimited input buffer that stores the messages as they come in. The nodes can consume at most one message per cycle. The code for the network is provided to you.
5. Each processor (trace reader) will have up to one outstanding memory request at a time. The processor sends its request to the cache and will wait until the cache responds with a DATA message using `send_DATA_proc()`. The caches will maintain coherence using the ring network and the directory.
6. We have provided a full simulator framework in C++. The only files you will need to modify are in the `coherence/` directory.
7. As a reminder, you will need to implement the agents and directory controllers in: `MSI_directory.cpp`, `MESI_agent.cpp/hpp`, `MOESIF_agent.cpp/hpp`
   - When a processor makes a request for the cache, the cache will find the cache entry then call the appropriate `process_cache_request()` for the protocol/agent being used. This function should look at a cache entry's state and determine which messages (if any) should be sent on the network, and which state to transition to. The function is already set up to call the appropriate `do_proc_XX()` helper functions that have been stubbed out for you.
   - When a request is encountered from the network, the cache will find the appropriate cache entry and call `process_ntwk_request()` for the appropriate protocol/agent. This function should look at a cache entry's state and determine which messages (if any) should be sent on the network, and which state to transition to. The function is already set up to call the appropriate `do_ntwk_XX()` helper functions that have been stubbed out for you.
   - When the directory controller receives a message, it should modify the state in the directory as needed and send any messages required to maintain the state. The directory controller will call the appropriate `PROTOCOL_tick()` function, which has been stubbed out for you.
   - You should not need to modify any of the header files provided to you and should not need any more state tracking variables than those already in the simulator infrastructure.

Note: In order to properly interface with the simulator infrastructure, do not change any class names or delete any functions.

3.2 Simulator Assumptions

1. All requests sent due to `do_proc_XX()` expect a response in the form of either DATA or ACK. In order to satisfy this, the directory will either initiate a cache-to-cache transfer or access memory to respond to the requesting node.
2. The input queues on each node are treated as FIFO in most cases. However, if a request cannot be processed in the current cycle (e.g. the corresponding entry is in a transient state and waiting for a reply from another node), pop this request and push it back to the end of the queue using `cycle_queue()`, this is done to avoid possible deadlocks.
3. If we are reading from memory, the other requests in the directory queue cannot be processed. The directory is stalled until the memory operation completes. In this case, we do not push the request to the back of the queue. In essence, we simply stall the input queue until the memory read completes. We do not need to wait for writebacks to complete (e.g. there is an infinite on-chip store buffer to DRAM). You can directly forward the data to the requester from the directory controller.

3.3 Directory Implementation

1. You will implement the basic Censier & Feautrier centralized directory scheme discussed in lecture. In order to reduce simulation runtime and reduce complexity, you do not need to explicitly build the complete directory for all of memory. You only need to keep directory entries for each active block (blocks that are present in at least one cache). The creation/initialization of agent entries for the cache and the directory entries for memory is handled for you. Coherence is maintained at the granularity of cache blocks.
2. In general, there is more than one way to implement each protocol. For this project, the reference implementations were made with simple logic, emphasizing cache-to-cache transfers, and running in minimal time, so please follow the instructions in this PDF instead of Wikipedia.
3. In order to simplify the simulator, all $-to-$ transfers will first have a node send DATA to the directory, then the directory controller will forward the data to the destination.
4. Silent upgrades: If a cache block is in E, the agent knows that the associated cache is the only cache that has the data. However, the directory may have made actions assuming the agent remained in E. This makes the "Silent Upgrade" discussed in class difficult to implement without substantial complexity. Therefore, we will instead implement a transient state in the Agent, state EM. When the agent in E sees a STORE request from the processor, it will send a GETX message to the directory and inform it that it wants to upgrade. The directory may not respond to this request immediately, meaning that the EM state may transition to the IM, SM, or FM state depending on the protocol. That said, if the agent is in the EM state and sees ACK from the directory, that means the directory has seen the E→M transition and the Agent can go to M and respond to the outstanding CPU request.
5. The directory may send messages like REQ_INVALID which should cause the agent to go to I or, if currently in a transient state, a transient state like IM.

6. You should not need to allocate or free any structures, the helper functions in `simulator/agent.h` and `send_Request` in the directory should handle all allocations and sending of messages. You just need to provide them with the right arguments. More documentation is provided in the source code.

3.4 Library of Messages

There are a number of messages that will be used in the simulator infrastructure. They can be found in `coherence/messages.h`

- `LOAD`: The CPU sends the cache agent this message to service a load in the trace. The cache will handle this by either sending the data to the CPU (if in M, S, E, F, or O) or by sending a GETS on the network.
- `STORE`: The CPU sends the cache agent this message to service a store in the trace. The cache will handle this by either sending the data to the CPU (if in M) or by sending a GETM (GETX if in E state) on the network.
- `GETS`: Get with intent to Share. This request informs the directory that the requester wants to read from the address.
- `GETM`: Get with intent to Modify. This request informs the directory that the requester wants to write to the address.
- `GETX`: Get with intent to Upgrade. This request informs the directory that the requester in E wants to upgrade to M. The state of the system may have changed by the time the directory sees this message, so this message may be seen in many states in the directory. The directory should respond to this with either ACK or with DATA depending on the state.
- `DATA`: The Directory may respond to a cache agent with this message when the cache sends a GET request. An agent will respond to the directory with this message when it sees a RECALL_GOTO_I or RECALL_GOTO_S.
- `REQ_INVALID`: The directory will request a cache agent to go to the I state with this message. The agent should either go to I or the appropriate transient state when it sees this message and send INVACK.
- `INVACK`: The cache agent should respond to REQ_INVALID with this
- `DATA_E`: The directory will respond to a cache agent with this when the cache should go to state E instead of state S since the cache is the exclusive sharer
- `RECALL_GOTO_I`: The directory uses this message as a mechanism to initiate cache to cache transfers
- `RECALL_GOTO_S`: The directory uses this message as a mechanism to initiate cache to cache transfers
- There are other messages defined in the `message_t` enum, but they are not used for this project.

3.5 Cache Agent State Transition Diagrams

See Appendix C for details on the Agents for each of the protocols. They should help you understand the variety of messages you should expect to send and receive for each protocol.

3.6 Simulator Parameters

The following command line parameters can be passed to the simulator:

- `-p`: The protocol that the simulator will be using (MSI, MESI, MOSI, and MOESIF).
- `-t`: The path of a directory containing the memory access traces for each core. For example, you could pass `-t traces/core_4` .
- `-n`: The number of processors. To determine this value, note the number of files in the individual trace directory determine how many cores you need to run the simulator with.
- `-h`: Print the usage information

3.6.1 Additional Parameters

The following parameters are available but not implemented and you should not modify them:

- `-c`: CPU Type: Only FICI In-Order CPUs are supported for now
- `-i`: The interconnect topology. For this project, only a simple RING network is supported
- `-l`: The link latency. For this project, each message only takes 1 cycle to take a step along the ring. In a real system, DATA messages could take many more cycles
- `-m`: The memory access latency. For this project, each memory request takes 100 cycles.

4 Statistics

We are concerned with the following statistics, which need to match exactly.

- `sim->clock`: Execution time in simulator cycles (The provided code manages this for you).
- `sim->cache_misses`: Counts when a CPU makes a request and the cache is in state I.
- `sim->cache_accesses`: The number of times the CPUs attempt to access the cache (The provided code manages this for you).
- `sim->silent_upgrades`: The number of silent upgrades for the MESI and MOESIF protocols.
- `sim->cache_to_cache_transfers`: The number of times data is provided by another cache and we do not have to go to memory.
- `sim->memory_reads`: The number of times we went to memory to access data (The provided code in the directory controller manages this stat for you but you still need to set up the directory controller correctly).
- `sim->memory_writes`: The number of times the directory received dirty data that needed to be written to memory.

Like previous projects, you can run `make validate` to compare with the reference outputs.

5 Implementation Details

You will need to modify the following files in `coherence/`:

- `directory/MSI_directory.cpp` - Where you will implement the directory controller for the MSI protocol.
- `agents/MESI_agent.cpp` - Where you will implement the per core agent for the MESI protocol.
- `agents/MOESIF_agent.cpp` - Where you will implement the per core agent for the MOESIF protocol.

You are encouraged to look at the pieces of the protocols that have been provided to help you understand how the agents/directories work.

Additionally, to avoid confusion from hanging simulators, we have added a check that exits the main simulation loop if the simulation has been running for 500,000 cycles. No simulation in this project should even approach that cycle count. If you see the message "is there a deadlock?" when running your code, please ensure that your code is not stuck in a transient state.

5.1 Docker Image

We have provided an Ubuntu 22.04 LTS Docker image for verifying that your code compiles and runs in the environment we expect — it is also useful if you do not have a Linux machine handy. To use it, install Docker (`https://docs.docker.com/get-docker/`) and extract the provided project tarball and run the `6290docker*` script in the project tarball corresponding to your OS. That should open an Ubuntu 22.04 `bash` shell where the project folder is mounted as a volume at the current directory in the container, allowing you to run whatever commands you want, such as `make`, `gdb`, `valgrind`, `./dirsim`, etc.

5.2 Important Note

The gradescope autograder for this assignment may use an optimized build (`make FAST=1`) due to the project runtime, ensure that your code validates locally with this flag. It is helpful to treat all warnings as errors. Ensure you use the docker container even if you didn't need it for past assignments.

6 Validation Requirements

You must run your simulator and debug it until the statistics from your solution **perfectly (100 percent)** match the statistics in the reference outputs for all test configurations. This requirement must be completed before you can proceed to Section 7 (Experiments).

You can run `make validate` to compare your output with the reference outputs. If you want to test only one protocol for all traces, you can use the `validate.sh` script directly and pass it a protocol name, like `./validate.sh MSI`.

We do not have a hard efficiency requirement in this assignment, but please make sure your simulator finishes a simulation for one of the provided traces in less than a minute or two (For reference, the TA solution finishes a simulation in a few seconds). Otherwise, it will be difficult for the TAs to verify your code produces matching outputs.

6.1 Debug Outputs

In contrast with past assignments, you can use a new function called `debug_printf()` to generate debug outputs. This means you won't need to add `#ifdef DEBUG` and `#endif` around your `printf()` statements. However, you still need to compile the simulator with `make clean && make DEBUG=1`. Then, you can find the debug outputs generated by the TA solution in `debug_outs/` directory of the assignment tarball and compare them with your debug outputs using a tool like `diff`. You are **not** required to match the TA debug outputs, but they may help with debugging.

You are strongly encouraged to use the debug outs as they will help you see exactly when the simulator state differs between the solution and your implementation. These debug outs are how we debugged and validated the simulator and should contain all the information required to find a bug.

Please do not submit code that always generates debug outputs or other print statements, as this will break the autograder and cause you not to match reference outputs.

6.2 Debugging

To debug, please use GDB (for segfaults) and Valgrind or Address Sanitizer (for memory corruption) as demonstrated in Appendix B. We also encourage comparing with the debug outputs (the tool `diff` is useful) before coming to office hours for help debugging your code.

7 Experiments

Once your simulator has been validated and matches the `ref_outs`, you will run the experiments in the `experiments/` directory included in the assignment tarball. Compare the performance of each experiment using the MSI, MESI, MOSI, and MOESIF protocols. You should run all experiments with 8 cores (-n 8).

Provide the execution time in cycles for each configuration. Using the simulator statistics and any other information from the course or your own research, explain why a certain protocol performs better for a certain experiment. Pick the best protocol for each experiment.

Ensure that the report is in a file named `<last name>_report.pdf`. Please submit a PDF and not other file types (no Microsoft Word documents, please).

7.1 Experiment core counts

You have been provide benchmarks to run for 8 cores:

- array_sum
  - array_block
  - array_stripe
- matrix_multiply (matmul)
  - matmul_row
  - matmul_col
- map_reduce

- pipeline_migrating

You are provided the source code for each of the benchmarks under `experiments/programs`, and your analysis will need to consider the behavior of each benchmark when explaining results. The top of each source file explains what the program does.

The traces for each benchmark contain the instruction fetch and load/store/atomic accesses for the worker threads.

For each benchmark, run experiments for 8 cores (-n 8), and pick the protocol (MSI, MESI, MOSI, or MOESIF) that takes the fewest total cycles. Look at the source and explain **why** the chosen protocol outperforms the other protocols.

For the array_sum benchmarks (array_block and array_stripe), explain why the performance differs even if the same protocol is chosen for both.

For the matrix multiplication benchmarks (matmul_rows and matmul_cols), explain why the performance differs even if the same protocol is chosen for both.

Your report should contain (at least) the following sections

1. Array Sum
   (a) Array Block
   (b) Array Stripe
2. Matrix Multiply
   (a) MatMul Rows
   (b) MatMul Cols
3. Pipeline
4. Map-Reduce

8 What to Submit to Gradescope

Please run `make submit` and submit the resulting tarball (`tar.gz`) to Gradescope. Do not submit the assignment PDF, traces, or other unnecessary files (using `make submit` avoids this). Running `make submit` will include PDFs in the project directory in the tarball, but **please make sure** that the command ran successfully and your experiments report PDF is present in your submission tarball. We will create a simple Gradescope autograder that will verify that your code compiles and matches the 10 million–branch `gcc` trace and a subset of the other reference traces. This autograder is a smoke test to check for any incompatibilities or big issues; it is not comprehensive.

Make sure you untar and check your submission tarball to ensure that all the required files are present in the tar before making your final submission to Gradescope!

9 Grading

You will be evaluated on the following criteria:

+0 :	You don't turn in anything (significant) by the deadline
+50 :	You turn in well commented significant code that compiles and runs but does **not** match the validation
+10 :	Your simulator **completely matches** the validation outputs for MSI
+10 :	Your simulator **completely matches** the validation outputs for MESI
+15 :	Your simulator **completely matches** the validation outputs for MOESIF
+15 :	Your experiments have been run and your report is of high quality

Appendix A - Plagiarism

We take academic plagiarism very seriously in this course. Any and all cases of plagiarism are reported to the Dean of Students. We use accepted forensic techniques to determine whether there is copying of a coding assignment. You may not do the following in addition to the Georgia Tech Honor Code:

- Copy/share code from/with your fellow classmates or from people who might have taken this course in prior semesters.
- Publish your assignments on public repositories, github, etc, that are accessible to other students.
- Submit an assignment with code or text from an AI assistant (e.g., ChatGPT).
- Look up solutions online. Trust us, we will know if you copy from online sources.
- Debug other people's code. You can ask for help with using debugging tools (Example: Hey XYZ, could you please show me how GDB works), but you may not ask or give help for debugging the cache simulator.
- You may not reuse any code from earlier courses even if you wrote it yourself. This means that you cannot reuse code that you might have written for this class if you had taken it in a prior semester. You must write all the code yourself and during this semester.

Using AI Assistants:

Anything you did not write in your assignment will be treated as an academic misconduct case. If you are unsure where the line is between collaborating with AI and copying AI, we recommend the following heuristics:

Heuristic 1: Never hit "Copy" within your conversation with an AI assistant. You can copy your own work into your own conversation, but do not copy anything from the conversation back into your assignment. Instead, use your interaction with the AI assistant as a learning experience, then let your assignment reflect your improved understanding.

Heuristic 2: Do not have your assignment and the AI agent open at the same time. Similar to the above, use your conversation with the AI as a learning experience, then close the interaction down, open your assignment, and let your assignment reflect your revised knowledge. This heuristic includes avoiding using AI directly integrated into your composition environment: just as you should not let a classmate write content or code directly into your submission, so also you should avoid using tools that directly add content to your submission.

Deviating from these heuristics does not automatically qualify as academic misconduct; however, following these heuristics essentially guarantees your collaboration will not cross the line into misconduct.

Appendix B - Helpful Tools

You might the following tools helpful:

- gdb: The GNU debugger will prove invaluable when you eventually run into that segfault. The Makefile provided to you enables the debug flag which generates the required symbol table for gdb by default.
  - You can invoke gdb for the with `gdb ./dirsim` and then run `run -i traces/<trace dir> <more dirsim args>` at the gdb prompt.
- Valgrind: Valgrind is really useful for detecting memory leaks. Use the following command to track all leaks and errors for the simulator:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
    ./dirsim -i traces/<trace dir> <more dirsim args>
```

- Address Sanitizer: This is similar to Valgrind, but often runs faster and can detect more issues. Compile your project with `make ASAN=1` to use it.

Appendix C - Cache Agent State Transition Diagrams

It is important to note that CPU side messages that do not change state are not shown in these diagrams. Messages that are sent by the cache agent (whether to the CPU or to the directory) are also not shown:

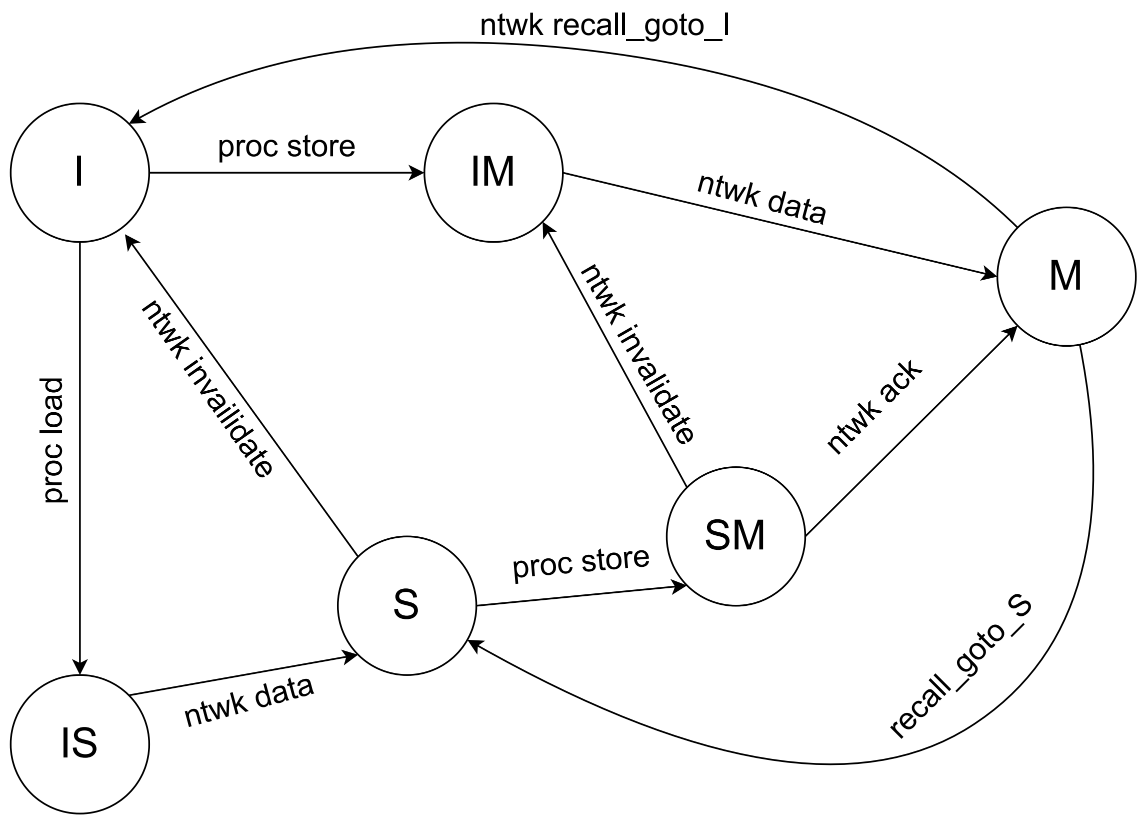

Figure 2: MSI Cache Agent State Transition Diagram

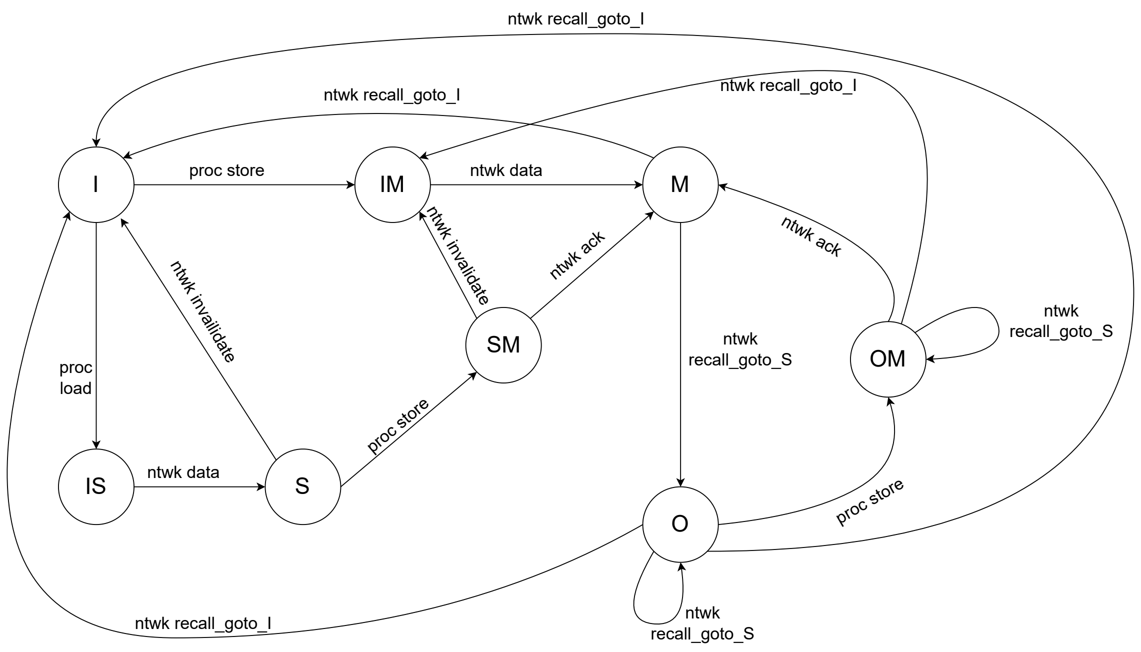

Figure 3: MOSI Cache Agent State Transition Diagram.

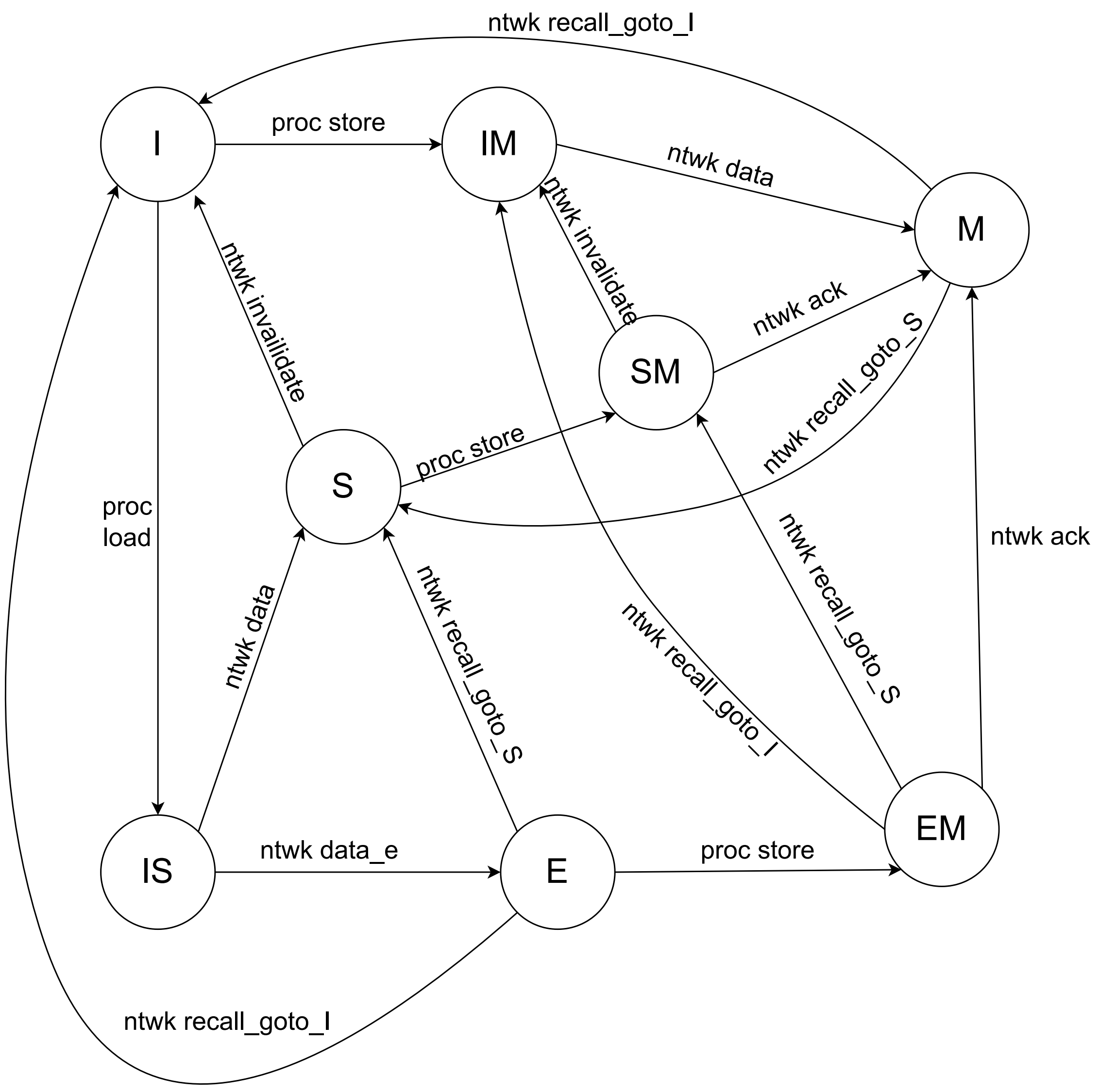

Figure 4: MESI Cache Agent State Transition Diagram.

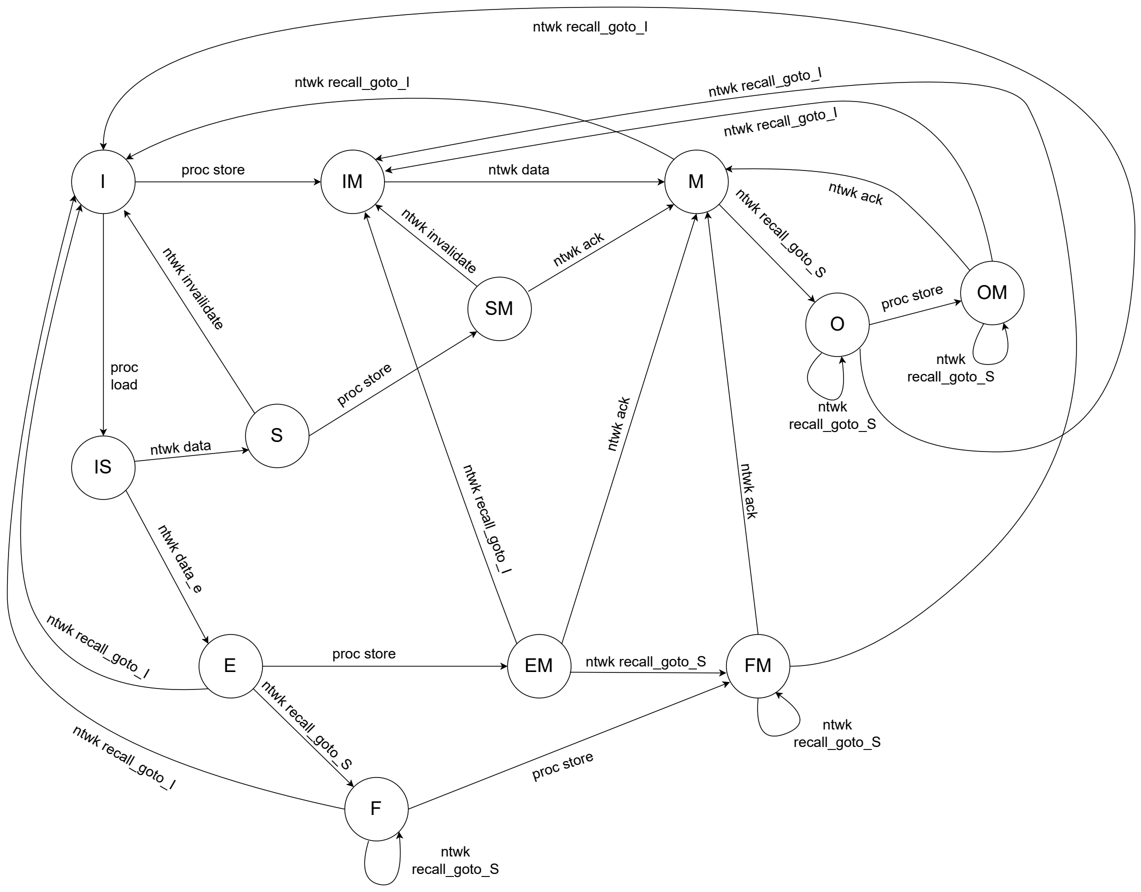

Figure 5: MOESIF Cache Agent State Transition Diagram.