18-640: Foundations of Computer Architecture

**Project 3**

Group NO.7 Luting Wang (lutingw) Wei Feng (wfeng1)

Part 1: Cache Replacement Policy

In this part, we are going to analyze two cache replacement policies: LRU (Least Recently Used) and LFU (Least Frequently Used). LRU replacement policy evicts the least recently used cache entry, while LRU replacement polity replaces the least frequently used cache entry. LRU is already in Gem5, and our task is to implement LFU and compare their results. We use PARSEC Benchmarks in FS mode and run two scripts with both policies. Our simulation results are as following:

1. Results

Table 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Blackschole*s* |  | LRU | LFU | X264*s* |  | LRU | LFU |
| SYSTEM.CPU.ICACHE.  TAGS.REPLACEMENTS  SYSTEM.CPU.CPI | 2.927503 | 3.415424 | SYSTEM.CPU.ICACHE.  TAGS.REPLACEMENTS  SYSTEM.CPU.CPI | 2.927606 | 3.415474 |
| 2.990673 | 3.666081 | 2.278857 | 2.799781 |
| 1.559660 | 2.148126 | 1.181013 | 1.515113 |
| 2.703057 | 3.190618 | 2.610446 | 3.273444 |
| SYSTEM.CPU.DCACHE.  TAGS.REPLACEMENTS | 1362613 | 1908057 |  | 1362613 | 1907939 |
| 31420 | 44083 | 48778 | 69934 |
| 4279 | 15349 | 23225 | 34495 |
| 17786 | 25886 | 16267 | 25752 |
| SYSTEM.CPU.CPI | 3010209 | 3468174 | SYSTEM.CPU.DCACHE.  TAGS.REPLACEMENTS  SYSTEM.CPU.CPI | 3010188 | 3468157 |
| 14211 | 21146 | 21166 | 33068 |
| 1861 | 2822 | 3735 | 7217 |
| 19551 | 24334 | 18738 | 24788 |

1. Questions
2. **Between LRU and LFU, which is better?**

I think LRU is better. Because CPI in LRU is smaller than that in LFU, and for ICACHE and DCACHE replacements, LRU has fewer replacement. So LRU has better performance.

1. **Analyze the results you obtained.**

From the statistics above, we can see that LRU performance is always better than LFU in both benchmarks. The reason is clear: when we perform LRU to evict cache entry, the temporal locality are considered which are frequent in most programs. If we refer to how recent a cache line is accessed instead of how many times a cache line is accessed, we can better exploit temporal locality and achieve better performance. And for cache replacement, we should notice that when we use full system mode to perform simulation, operating system, kernels and all other overheads are simulated. So if there is a context switch, LFU will replaces most of the data in cache, then load and warmup them again when the benchmark resume. So LFU has more cache replacement.

1. **Which of the two you think is easier to implement in hardware? Give reasons.**

I think LFU may be easier to implement in hardware. To implement LFU, we need to add counters to each block to record how many times it used. On the other hand, to implement LRU, we need a stack (FIFO). It has recent used block in the head, and the replacement happens in the tail. Although counters would occupy more memory, stack is more complicated to implement, so LFU is easier to implement from hardware view.

Part 2: Cache coherence and SLICC

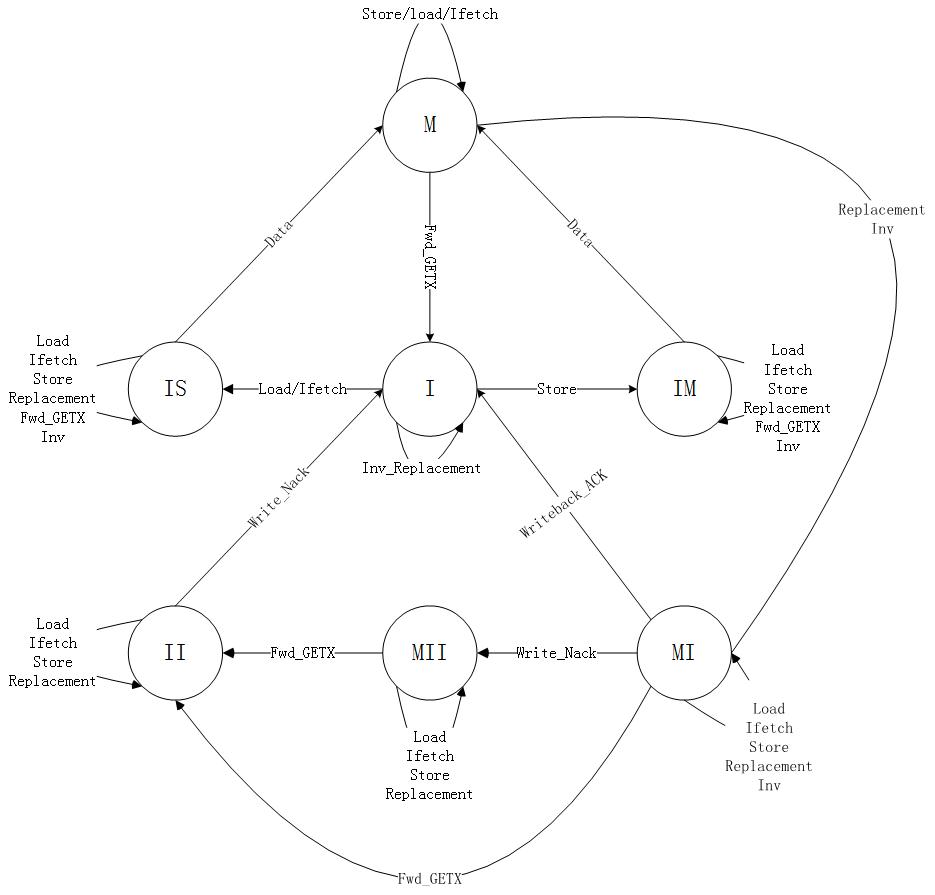


Figure 1 Cache Controller State Machine In MI Protocol

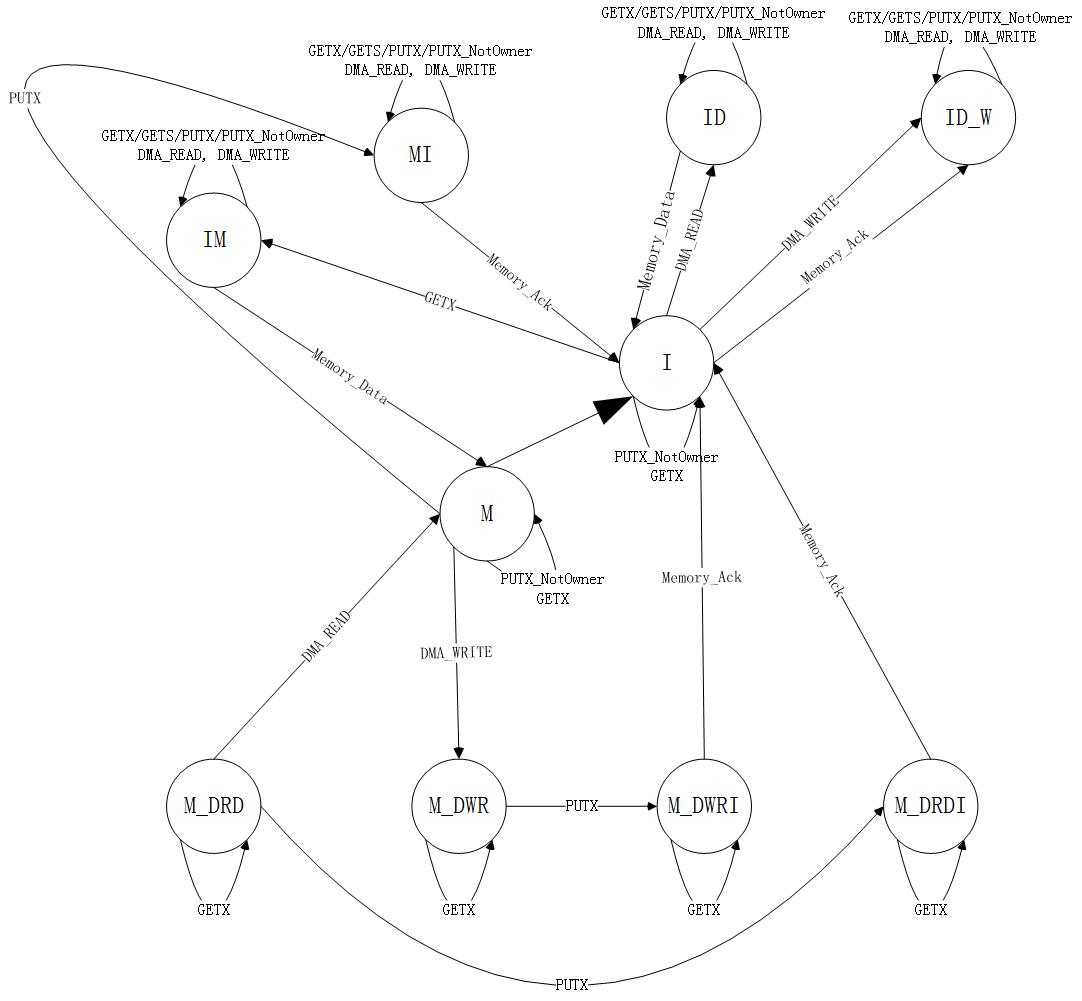


Figure 2 Directory Controller State Machine In MI Protocol

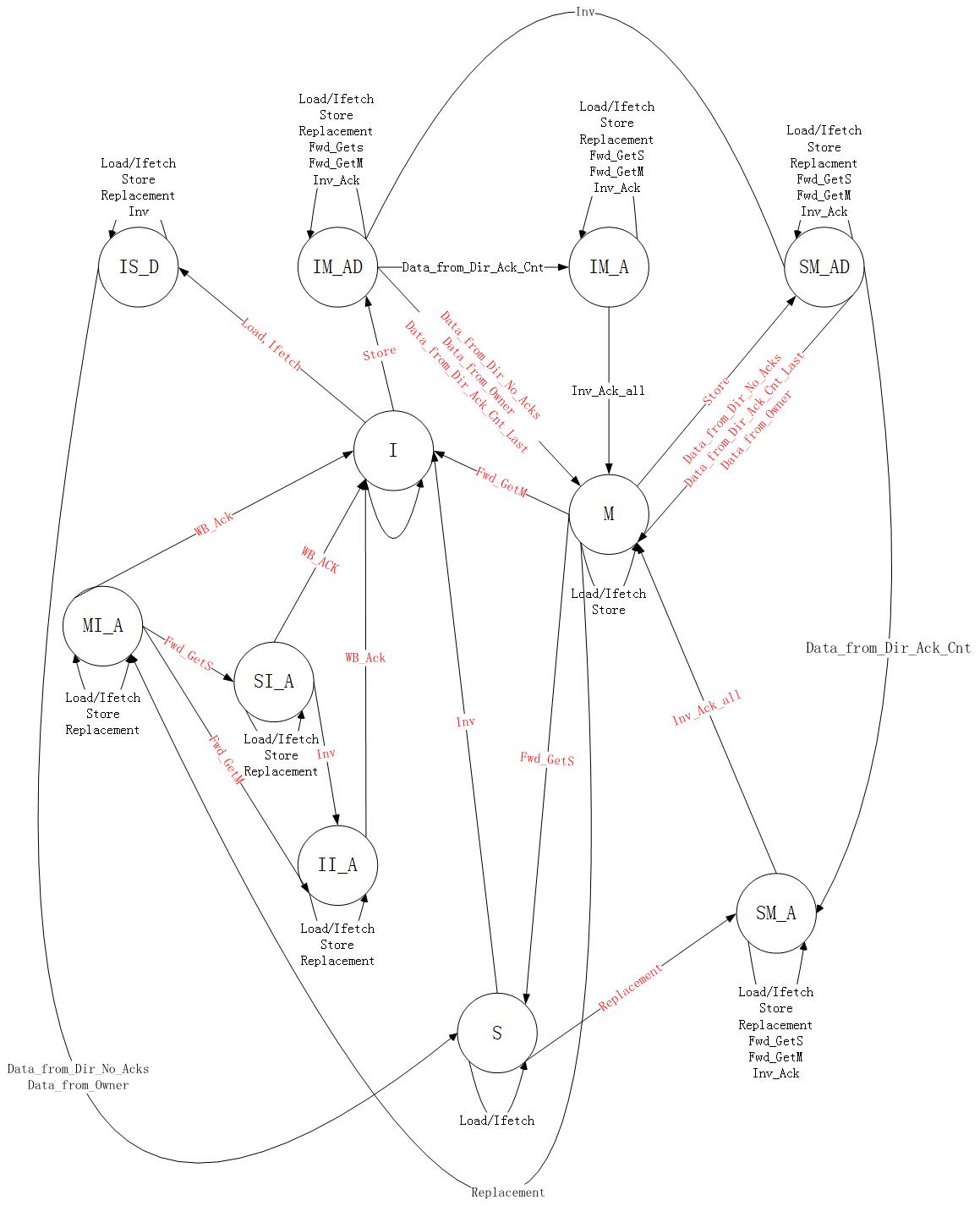


Figure 3 Cache Controller State Machine In MSI Protocol

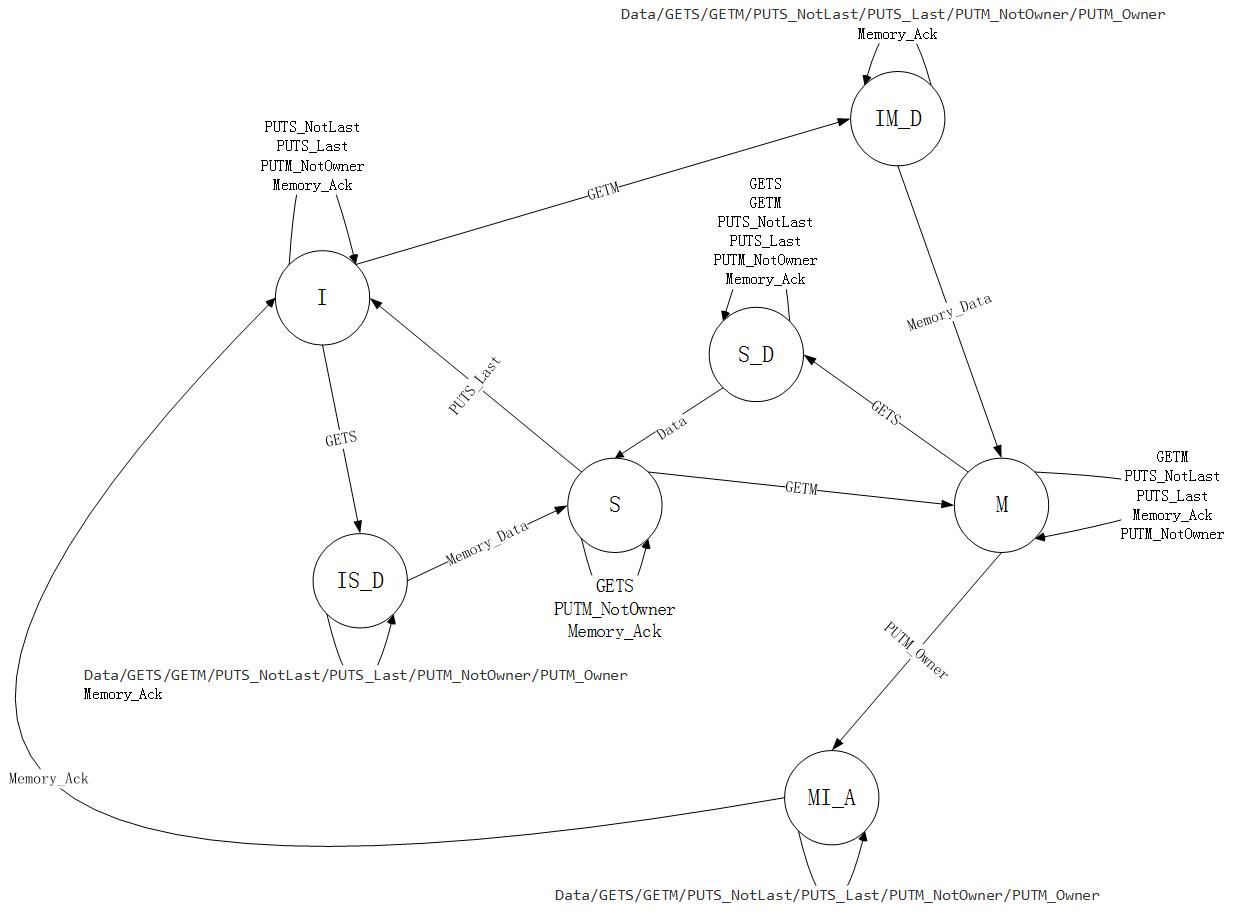


Figure 4 Directory Controller State Machine In MSI Protocol

1. Questions
2. **Why are transient states necessary?**

Because transitions between stable states are not atomic, so the transient states are necessary. For example, consider the MSI protocol, assume that the block copy is in state S when the processor issues a write to that block. Further, assume that before the BusUpgr request has been launched on the bus, a remote cache has launched a BusUpgr request for the same block on the bus. This upgrade invalidates the local block copy. More importantly, the BusUpgr request issued by the local cache is no longer relevant as the correct request to launch now is a BusRdX Request.

1. **Why does a coherence protocol use stalls?**

Using Stalls in a coherence protocol is a simple way to handle events that can’t be processed because of the cache block being in a transient state.

1. **What is deadlock and how can we avoid it?**

A deadlock is a situation in which two or more competing processors are each waiting for the other to finish, and thus neither ever does. In gem5 SLICC, a deadlock will occur when a transition statement is left blank, this will make SLICC to analyze the same input port repeatedly and it will lead to a deadlock. Typically, deadlocks can be avoided by an entity that manages the allocation of resources. In gem5 SLICC, deadlocks are avoided by the Sequencer. By periodically waking up and scanning though the m\_writeRequestTable and m\_readRequestTable tables, Sequencer makes sure that Cache hierarchy is making progress in servicing the memory requests that have been issued. When the Sequencer using a threshold parameter (m\_deadlock\_threshold) to finds out that there is no progress in the Cache hierarchy and reports a possible deadlock. Then, it wakes up and scanS the tables which hold currently outstanding requests.

Part 3: Cache coherence - Protocol Analysis

1. Results

In this part, we run two benchmarks to analyze directory-based protocols and snooping-based protocols. To evaluate their performance, we choose CPI and icacheStallCycles to analyze, because we think we can evaluate cache’s coherence performance by comparing their CPI and stalled cycles for instruction cache. The results are shown below:

Table 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Jpeg-decode | | quicksort | |
|  | CPI | icacheStallCycle | CPI | icacheStallCycle |
| Snooping | 0.618040 | 781283 | 1.144601 | 3550832 |
| Directory | 2.491418 | 10210368 | 3.262012 | 40999691 |

As what shown in the table above, CPI is lower in snooping-based protocols compared with directory-based protocols. And the reason may be there is less stalled cycles for instruction fetch process, as icacheStallCycle illustrated. In a snoop-based system, a dirty miss can be satisfied directly, since the read request is transmitted directly to the responder that has the dirty data. In a directory implementation, however, the request is first sent to the directory and then forwarded to the current owner of the line; this results in an additional traversal of the processor/memory interconnect and increases latency [1].

1. Questions
2. **Are the results intuitive? Give reasons.**

Yes, it is intuitive. CPI and icacheStallCycle in directory-based protocols is larger because there is longer latency of cache misses and upgrades, as requests are always sent first to the home node. So it may involve up to four interconnection network traversals to carry out BusRd, BusUpgr and BusRdx requests, while snooping-based protocol only need two hops when there is a dirty miss.

1. **When is snooping-based desirable over directory-based, and vice-versa?**

There is never a situation that snooping-based protocol is always over directory-based protocol, or vice-versa.

Snooping-based protocol is desirable over directory-based protocol when there are fewer processors in the system. Also, if dirty misses happen frequently, snooping-based protocol is better because such misses can be satisfied directly from the remote cache.

On the other hand, directory-based protocol is better when there are large number of processors in the system, because address commands will be sent to the directory first and are forwarded to remote processors only when necessary, so as the number of nodes scale up, the memory bandwidth will scale up proportionally. Also, latency for misses that are satisfied from memory can be significantly reduced, since the memory bank can respond with non-speculative data as soon as it has checked the directory [1].

1. **What are the different** **topologies you can simulate in** **gem5? What are their advantages? Which one did you simulate i.e. the default topology?**

There are five different topologies provided in gem5: Crossbar, Mesh, MeshDirCorners, Pt2Pt2 and Torus, and the default one is crossbar [2].

Crossbar: Every node is connected to every other through a switch. It enables concurrent transfers to non-conflicting destinations, and is cost-effective for small number of nodes.

Mesh: Every node is connected to its neighbors, and message is propagated along a path by hopping from node to node until it reaches its destination. It is easy to layout on-chip, with O(N) cost is and O(sqrt(N)) average latency.

MeshDirCorners: It is a specific case of mesh topology. It requires the number of directories to be equal to four. The main advantage is that it is less complex to implement.

Pt2Pt2: Every node is connected to every other with direct/isolated links. It has the lowest contention but highest cost.

Torus: It is an extension of the mesh architecture. It avoids large path distance between edge nodes by adding additional links at the edge.

[1] Shen J P, Lipasti M H. Modern processor design: fundamentals of superscalar processors[M]. Waveland Press, 2013.

[2] http://www.m5sim.org/Interconnection\_Network