ECE 585
Simulated LLC Final Report
Team 4

Portland State University

Maseeh College of Engineering & Computer Science

Department of Electrical & Computer Engineering

Authors:

Kane-Pardy, Chris

Melli, Ameer

Monti, Caleb

December 10, 2024

ECE 585

Microprocessor System Design

Contents

1 Introduction

This project simulates a last-level cache (LLC) in a multi-level, multiprocessor cache system, designed to visualize and analyze data organization, movement, and coherence within a shared memory architecture. The simulation also models inclusivity via coherence messages sent to other caches, as well as the higher-level cache for the processor our LLC operates on. Key behaviors such as hits, misses, evictions, and inter-cache communication are also tracked and modeled through print statement bus operations and counter functions.

The LLC simulation features two operating modes: Silent Mode (mode 0) and Normal Mode (mode 1). Silent Mode focuses on essential cache statistics, reporting the number of cache reads, writes, hits, misses, and the overall cache hit ratio, along with responses to specific operations. Normal Mode builds on this, additionally displaying bus operations with snoop results, and messages to the higher-level cache to maintain inclusivity. By default, the program operates in Silent Mode (`-m 0`) but can be switched to Normal Mode by passing the argument `-m 1` at runtime.

The simulation requires a trace file to operate, which can be specified with the `-f <filename>` argument. If no file is provided or the specified file is invalid, the program uses a default trace file showcasing basic functionality. Should neither the provided nor the default file be available, the program terminates with an error.

By simulating the behavior and interactions of a 16 MB, 16-way set associative LLC using the MESI protocol and a pseudo-LRU replacement policy, this project imparts a more profound and personal understanding of the inner workings of a cache. Additionally, the simulation offers an opportunity to practice system modeling, testing, and documentation, forming a foundation for possible practical implementation in real-world multi-processor systems design.

2 Assumptions

Cache Parameters

The following are all constructed as either functions or defines due to the assumption they are meant to be semi-portable for Caches of a different:

- Cache Capacity
- Line Size
- Address Length
- Associativity
- Replacement Policy
- Inclusivity/Exclusivity

Cache Functionality

- Memory references do not cross cache line boundaries and are aligned on 64-byte "chunks" automatically when read from DRAM
- Cache line fills do not use critical word first or early restart, and are atomic (no partial cache line fills)
- There is a fixed latency for all bus operations (No delay/waiting for operations or requests (not including printouts))
- The only supported cache operations are (n = 0 - 9 and not including 7), where n is read from the trace file of the form: (<operation> <address>), followed by a newline
- Snarfing is not allowed
- The address length (`ADDRESS_SIZE`) determines the maximum addressable memory range, and if all addresses used in the simulation fall within this range, else an error will occur and the code will terminate.
- Cache lines are uniquely identified by their tag and index, with no aliasing.
- Cache parameters aren't meant to change during runtime

Higher-Level Cache

- The only processor requests we'll ever see are misses in the higher-level cache
- The higher-level cache keeps track of state transitions and properly adheres to a coherence protocol
- The higher-level cache has a total capacity smaller than the LLC for inclusivity to be possible
- The higher-level cache correctly handles everything through inclusivity messages (e.g., INVALIDATELINE or EVICTLINE)

Coherence

- Our LLC can silently evict (not issue a bus operation or print statement (other than the writeback of a modified line being locally evicted)) without disrupting coherence
- The LLC is responsible for all writebacks to DRAM
- All addresses provided to this cache simulation will adhere to the same methodology for reporting HIT, HITM, and NOHIT for snoop requests
- This cache doesn't have distinct Flush and FlushWB commands for forwarding like MESIF protocol caches do, meaning every write-back must go through DRAM

PLRU

- The PLRU reads in from far Left (0) to far Right (14) and searches breadth-first
- On access a 0 signifies left-child and 1 signifies right-child, for eviction, the directions are reversed

Multiprocessor Functionality

- The simulation does not model concurrent operations, interleaving of cache requests, or arbitration
- There is no requirement for any bus contention modeling

3 Interface

The simulated cache model utilizes two primary interfaces for interaction. The first is the shared bus, through which the cache can perform the bus operations *BusRd*, *BusRdX*, *FlushWB*, and *BusUpgr*. This shared bus enables the cache to send and view snoop results to and from other caches (HIT, HITM, NOHIT). Its purpose is to maintain coherence and data consistency across caches sharing access to the systems main memory. Coherence is achieved by ensuring that no cache or processor can read or modify data without all caches holding a copy of the same data address being updated, thereby preserving consistency.

The second interface is the communication link with the higher-level cache. This model maintains inclusivity, requiring that the entire contents of the higher-level cache must also reside within this cache. Consequently, if a cache line is invalidated or evicted in the LLC, the same must occur in the higher-level cache. In the simulation, this communication is represented by printing messages that reflect the cache's actions—such as requesting, sending, or removing data—in response to specific events.

4 Internal Design

To parse the trace file and update the cache contents properly, it was necessary to split the requirements into multiple functions and call them as needed for each operation. The body of the cache code resides in main and involves a looping function that reads each line of the trace file sequentially. The first function called parses the byte select, tag, index, and operation from the given number and address. Then, each operation triggers the necessary functions to get the exact required result using case statements. By compartmentalizing the code as we did, we're able to ensure deterministic execution.

Operations that induce a processor operation will then run functions to determine if the requested address is already in the cache (a hit or miss). This is done by first checking for valid ways in the set extracted from the address. Each valid way is then checked resulting in a hit and updating the PLRU, or performing bus operations to acquire the address from DRAM while maintaining cache coherence by viewing snoop results. Once the operation is completed, the specific way in the set that the address was accessed will then be reflected as the most recent within the PLRU system (i.e. any given way in a set will give back the same address that was most recently stored to it), successfully mimicking a memory. The function will also update the MESI bits for each stored address to reflect the operation that occurred using case analysis. Regardless of hit or miss, messages are sent to the higher-level cache to mimic inclusivity operations.

Operations that emulate a snooped bus operation will echo the snoop result corresponding to the MESI state of the line in our LLC, and trigger any further bus operations necessary (e.g. a BusUpgr or BusRdX). Since data was not accessed, the PLRU is not updated, but the MESI bits and inclusivity statements are updated and sent as needed depending on the operation.

The remaining operations, Clear and Print, go through the entire cache and will either set the contents to the INVALID state or print all ways and set PLRUs not in the INVALID state. While the Print function does not directly interface with the data lines and therefore does not update any values or states; the Clear function will need to empty the cache, as well as send messages to the higher level cache to invalidate its contents. Therefore, a function was implemented to also invalidate or evict each of the lines in the higher level cache depending on its current MESI state.

The code also integrates modular calculations for cache parameters such as the tag, index, and byte-select bits, which are derived dynamically based on user-defined cache configurations. This makes the implementation flexible and adaptable for caches with different sizes, associativities, or line sizes. Memory management is handled dynamically for structures such as PLRU arrays and ways within sets, ensuring scalability. Error handling is incorporated for cases where memory allocation fails, with appropriate debug messages for clarity.

In the event of issues or errors, there is a debug flag that can be used at compile time to generate a thorough log of cache operations, including intermediate calculations, MESI updates, and bus operations. This mode also forces the program into Normal Mode, maximizing output for easier error tracking and resolution.

5 Simulation Results

The simulation was validated using a series of structured test cases to verify the successful operations of individual functions and the system as a whole. This section discusses the outcomes of key tests, covering the operational modes, associativity, PLRU functionality, write-allocate behavior, the MESI protocol, cache coherence, inclusivity, and cache statistics.

Operational Modes

Purpose: To verify that **Debug**, **Normal**, and **Silent** modes operate correctly and produce expected outputs.

Debug Mode

Test Purpose: Validate comprehensive internal logs under `-DDEBUG`.
Methodology:

- Compiled with `-DDEBUG` to activate debug logs.
- Ran the program with a given test file to observe cache actions, bus operations, snoop responses, state transitions, evictions, and replacement decisions.
- Validated internal logging output alongside summarized statistics.

Results: Debug mode produced comprehensive logs as expected, including all cache actions, bus operations, snoop responses, state transitions, decision-making (e.g., evictions, misses, replacements), fprintf debug messages, and the overall reads, writes, hits, misses, and the hit ratio.

Normal Mode

Test Purpose: Validate mode switching using `-m 1` and confirm it logs real-time activity.
Methodology:

- Ran the program with `-m 1`.
- Provided a trace file with cache operations (reads, writes, bus transactions, snoop requests).
- Checked for detailed real-time logs, bus operations, and summarized statistics.

Results: Normal Mode showed real-time activity and the output provided real-time logs for all the cache actions, bus operations, state transitions, and snoop results. Along with a summarized output of the cache statistics for the overall reads, writes, hits, misses, and the hit ratio.

Silent Mode

Test Purpose: Validate that intermediate logs are suppressed in Silent Mode (`-m 0`) and only summarized statistics are displayed.
Methodology:

- Ran the same trace that was used for the Normal Mode test without any arguments to use the default Silent Mode.
- Also explicitly ran with `-m 0`.

Results: Only a summary of the cache statistics (and n = 9 printouts) was present during the silent mode run.

Associativity Testing

Purpose: To ensure we could support 16-way associativity where 16 tags can be stored at the same index before eviction, and that our design also met cache capacity constraints.

Test Scenarios:

1. `4_1_1.din`: Used trace file to load 16 unique memory addresses into a single set index. Observed the behavior using `n=9`.
2. `4_2_1.din`: A 17th address is added to a full set to trigger an eviction.
3. `4_3_1.din`: This trace contained 256 Ki unique memory addresses. Ensures that addresses with identical set indexes but different tags are stored in separate ways by completely filling the cache.

Results:

1. 16 unique cache lines were successfully stored in a single set with no evictions.
2. When the 17th address was added, eviction occurred as expected.
3. The cache's associativity and address mapping (tag + index bits) worked as intended, resulting in 256 Ki misses.

PLRU Functionality

Purpose: To validate the pseudo-LRU replacement policy's placement and eviction under processor and bus operations.

Tests Conducted:

1. `5_1.din`: Tested initial PLRU fill and update behavior without eviction under 16-way associativity.
2. `5_2.din`: Tested eviction by filling all cache lines in a set, accessing a subset of lines, and evicting the least recently used (LRU) entry.

Results:

1. The PLRU mechanism tracked accesses and updated its vector to accurately mark the most recently used way.
2. Evicted the least recently used way, corresponding with our manual calculation of the victim way using a PLRU tree.

Write-Allocate Testing

Purpose: To verify that our LLC follows the write-allocate policy and MESI transitions.

Test Scenarios:

1. `6_1.din`: Tested a trace that resulted in a write miss.
2. `6_2.din`: Tested multiple writes to the same address following an initial write miss.

Results:

1. Observed cache-line allocation and a subsequent load from the next level of memory. Verified that MESI transitioned to **MODIFIED** after the first write.
2. Subsequent writes to a line in the modified state resulted in cache hits without requiring additional bus transactions.

MESI Testing

Purpose: To verify that MESI states update properly to all possible bus operations (Case Analysis).

Test Scenarios:

1. `7_1.din`: Tested all cases in **SHARED** (PrRd, PrWr, BusRd, BusUpgr)
2. `7_2.din`: Tested all cases in **EXCLUSIVE** (PrRd, PrWr, BusRd, BusRdX)
3. `7_3.din`: Tested all cases in **MODIFIED** (PrRd, PrWr, BusRd, BusUpgr)
4. `7_4.din`: Tested all cases in **INVALID** (PrRd, PrWr, BusRd, BusUpgr, BusRdX)

Results:

1. PrRd: Remains in **SHARED**
   PrWr: A BusUpgr is issued and the line transitions to **MODIFIED**
   BusRd: The line remains in **SHARED**
   BusUpgr: The line transitions to **INVALID**
2. PrRd: Remains in **EXCLUSIVE**
   PrWr: The line transitions to **MODIFIED**
   BusRd: The line transitions to **SHARED**
   BusRdX: The line transitions to **INVALID**
3. PrRd: Remains in **MODIFIED**
   PrWr: The line remains to **MODIFIED**
   BusRd: A FlushWB is issued and the line transitions to **SHARED**
   BusRdX: The line transitions to **INVALID**
4. PrRd: Remains in **SHARED** or **EXCLUSIVE** accurately depending on the received snoop result
   PrWr: The line transitions to **MODIFIED**
   BusRd/BusUpgr/BusRdX: The line remains to **INVALID**

Cache Coherence Testing

Purpose: These test scenarios aim to comprehensively validate the cache's coherence under various operations, ensuring compliance with MESI protocol and proper handling of bus operations.

Test Scenarios:

1. `8_1.din`: Simulate a cache miss for a PrRd, then observe the results to a HIT, HITM, and NOHIT snoop result respectively.
2. `8_2.din`: Simulate a PrWr HIT to a line in **EXCLUSIVE** and **SHARED**, and a PrWr MISS to a line in **INVALID** for both HIT/NOHIT and HITM snoop results.
3. `8_3.din`: Simulate a snooped BusRd for a line in the **SHARED**, **EXCLUSIVE**, **MODIFIED**, and **INVALID** state.
4. `8_4.din`: Observe a FlushWB from another cache for a line that exists in our LLC.
5. `8_5.din`: Simulated a snooped BusRdX/BusUpgr for a line in every state.
6. `8_6.din`: Run the Clear Cache function for both an unmodified (**E** or **S**) and **MODIFIED** line.

Results:

1. The missed PrRd generated a BusRd, then for a snooped-
   HIT: The line is read into the **SHARED** state
   HITM: After a FlushWB, the line is read into the **SHARED** state
   NOHIT: The line is read into the **EXCLUSIVE** state
2. PrWR HIT in **EXCLUSIVE** leads to the line moving into **MODIFIED**
   PrWR HIT in **SHARED** leads to a BusUpgr being issued and the line moving into **MODIFIED**
   PrWR MISS in **INVALID** with a snooped HIT or NOHIT leads the issuing

of a BusRdX and the line moving into **MODIFIED**
PrWR MISS in **INVALID** with a snooped HITM leads the issuing of a BusRdX, FlushWB, and the line moving into **MODIFIED**

3. For the line in **S**, a HIT snoop result is sent
   For the line in **E**, a HIT snoop result is sent and the line transitions to **S**
   For the line in **M**, a HITM snoop result is sent, a FlushWB is issued, and the line transitions to **S**
   For the line in **I**, the LLC sends a NOHIT snoop result
4. The FlushWB is observed with no impact or responses sent.
5. All lines transition to **INVALID**, and echo the accurate snoop result flawlessly.
6. Unmodified Line: The line moves to **INVALID**
   MODIFIED Line: A FlushWB is issued and the line moves to **INVALID**

Write-Once Policy

Purpose: To ensure that the first write to a cache line implements the write-once behavior correctly, transitioning the line to the **MODIFIED** state and correctly handling subsequent accesses.

Test Scenarios:

1. `10_1.din`: Issue a PrWr to any cache line, then a subsequent BusRd.

Results:

1. The LLC transitioned the first write operation to the **MODIFIED** state as expected. Upon a BusRd to the same address, the LLC successfully issued a `GETLINE`, a FlushWB, and transitioned the cache line to the **SHARED** state.

Inclusivity Testing

Purpose: To confirm that the LLC maintains inclusivity by sending invalidation and eviction messages to higher-level caches when necessary.

Test Scenarios:

1. `9_1.din`: Tested `SENDLINE` for PrRd and PrWr.
2. `9_2.din`: Tested `GETLINE` for BusRd.
3. `9_3_1.din`: Tested `INVALIDATE` for Collision Misses.
4. `9_3_2.din`: Tested `SENDLINE` for Cache Clear.
5. `9_4.din`: Tested `SENDLINE` for Collision Misses and Cache Clear.

Results: The cache successfully sent all expected inclusivity messages. The operations worked as expected:

1. `SENDLINE` ensured the cache could receive data after a miss on both PrRd and PrWR.
2. `GETLINE` correctly fetched the data from a **MODIFIED** state for a FlushWB.
3. `INVALIDATELINE` was sent correctly during Cache Clear and Collision Misses.
4. `EVICTLINE` sent the correct data back to DRAM while transitioning the cache state to **INVALID** during both Collision Misses and Cache Clears.

Cache Statistics Testing

Purpose: To ensure that the overall reads, writes, hits, misses, and the calculated hit ratio were accurately tracked and displayed at the end of the simulation.

Test Scenarios: Cache statistics were tested using the trace file `11_1.din`, which included:

7 reads
5 writes
4 hits
8 misses
Hit Ratio: 0.33333

Methodology:
1. The simulation was executed using the trace file `11_1.din`.
2. Observed the program's output to ensure that overall statistics (reads, writes, hits, misses, and hit ratio) were calculated and displayed accurately.
3. A shell script was developed to automate validation by comparing expected statistics with the actual output generated by the simulation. The first test checked if the number of reads and writes was accurate. The second test verified that the number of hits and misses matched the expected values. The third test checked if the calculated hit ratio was computed as expected based on the input statistics.

Results:
The results from the simulation matched the expected values:
Reads and writes were accurately tracked
Hits and misses corresponded correctly to the expected results based on `11_1.din`
The calculated hit ratio of 0.33333 was computed correctly.

6 Conclusion

In this project, we successfully designed, implemented, and verified a simulation of a last-level cache (LLC) with a focus on the given key design principles such as associativity, cache coherence, inclusivity, and MESI state transitions. The simulation was implemented using modular design principles, with well-separated functionalities for cache modes, bus transactions, and cache coherence policies. The primary goal was to ensure that the simulation adhered to all provided specifications while allowing flexibility through configurable parameters and multiple testing modes.

The simulation results confirmed that all design and functional requirements were met, including proper cache behavior under varying operational modes, accurate MESI state handling, coherent bus communication, and correct cache statistics reporting. The modular design of the simulation code allowed efficient testing, easy reconfiguration for other system parameters, and a clear path for future enhancements if this project were to continue.

The results, combined with systematic testing and validation, indicate that the simulation provides a reliable representation of a last-level cache's behavior under multi-processor shared memory conditions. Moving forward, this simulation can be extended to incorporate more complex memory hierarchies, different replacement or coherence policies, and different cache parameters. The flexibility and modularity of this simulation lay the groundwork for these future extensions. Overall, we feel we've successfully implemented a functional simulation that meets all design requirements and accurately displays the behavior of an LLC with the given parameters