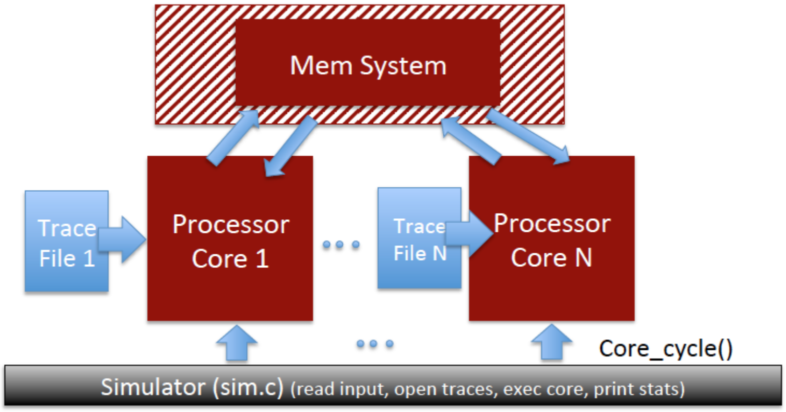
CS 4290 / CS 6290 / ECE 4100 / ECE 6100
Advanced Computer Architecture

Lab 4: CMP Memory System Design (10 (+ 2) Pts)

Parts A + B + C Due: Wednesday, November 22, 2023 (11:55 pm)
Part D (+ E) Due: Friday, December 1, 2023 (11:55 pm)



This is an individual assignment. You can discuss this assignment with other classmates but you should code your assignment individually. *You are NOT allowed to see the code of (or show your code to) other students or of past submissions that may be available online.* We will be using software that detects code similarity. If you think you collaborated closely enough with another student that could lead to more code similarities than would normally be expected, please declare it upfront, at time of submission (as a comment in your Canvas submission). *Please note that such declaration is not granting you an excuse for copying code.*

We will provide you with reference execution times for every part. If your code takes way more time (e.g., 5x) to execute than the reference time, you may be doing something very inefficiently and may want to reconsider your implementation. You can time your experiments using the *time* command. We will overprovision the time limit on the autograder, but if your execution time is drastically higher, your submission may be timing out so you won't be able to get autograder results. However, we won't grade you on how long your code takes to run. Our final grading will remove execution time restrictions, so long runtimes won't ultimately penalize you.

OBJECTIVE

The objective of the fourth (and last) programming assignment is to build and evaluate a (performance-only) multi-level cache simulator with DRAM-based main memory. The system will then be extended to incorporate multiple cores, where each core has a private L1 (separate Instruction and Data) cache, and a shared L2 cache. Misses and writebacks from the shared L2 cache are serviced by a DRAM-based main memory consisting of 16 banks and per-bank row buffers.

PROBLEM DESCRIPTION

The figure above shows the multicore system with the memory hierarchy that you will build as part of this lab. We will build the simulator for this system in two phases. The first phase (A, B, C) is for a single-core system and the second phase (D, E, F) extends the system to support multiple cores.

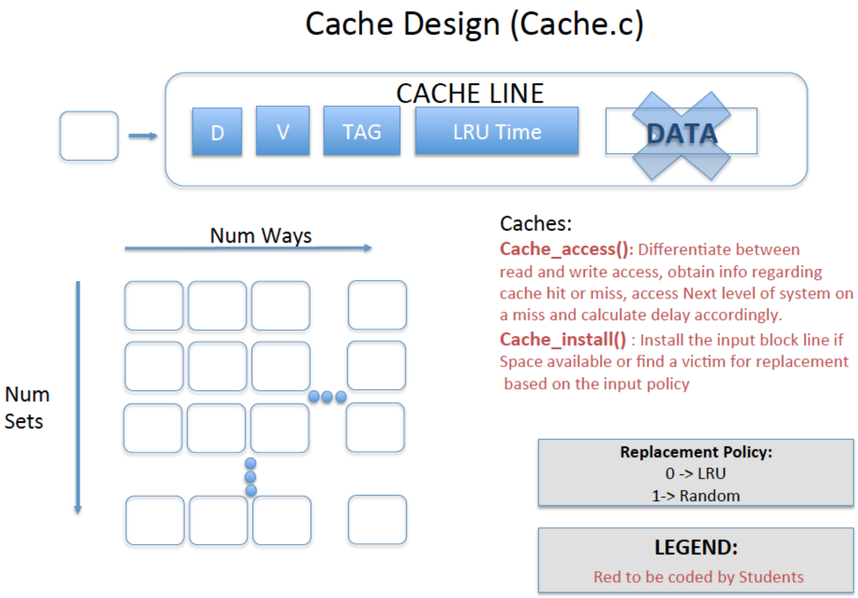
Tip: The complexity of the system you must implement in this lab is lower than Lab 3. However, you are given a more loosely defined code structure. Hence, you should plan to invest more time upfront, designing your code's skeleton, before directly jumping into coding.

Part A: Design a Standalone Cache Module (3 points)

Reference execution time: 9-10 sec per trace

In this part, you will build a cache model and estimate the cache miss ratio. You need to implement your own data structure for the cache (name it **Cache**) in cache.{cpp, h} and leave all other files untouched (except for run.sh). The cache functions must be written to work for different associativity, cache line size, cache size, and replacement policies. Refer to Appendix A for more details on the cache implementation.

Note: Remember that the terms "cache line" and "cache block" are synonymous.



Cache Design (Cache.c)

Caches:
Cache_access(): Differentiate between read and write access, obtain info regarding cache hit or miss, access Next level of system on a miss and calculate delay accordingly.
Cache_install() : Install the input block line if Space available or find a victim for replacement based on the input policy

Replacement Policy:
0 -> LRU
1-> Random

LEGEND:
Red to be coded by Students

As we are only interested in cache hit/miss information for performance evaluation, we will not be storing data values in our cache. We provide you with traces of 100M instructions from three SPEC2006 benchmarks: bzip2, lbm, and libq.

Each trace record consists of 4 bytes of instruction PC, 1 byte of instruction type (0: ALU, 1:LOAD, 2:STORE) and 4 bytes of virtual address for LD/ST instructions. We are providing you with the trace reader and the ability to change cache parameters from the command line. The trace reader calls the memory system for LD and ST instructions and the function in the memory system in turn calls the **cache_access** function for the DCACHE. If the line is not found in the DCACHE, the memsys function calls the **cache_install** function for the DCACHE, which in turn calls **cache_find_victim**. You should implement an **LRU replacement policy**. In the end, the memory system also calls the cache_print_stats function for the DCACHE. The cache print stats functions are already written for you.

Your objective is to implement three functions in cache.cpp:

1. **cache_access** – if the line is present in the cache, return HIT.
2. **cache_install –** install the line in the cache and track the evicted line.
3. **cache_find_victim –** find the victim to be evicted.

Refer to Appendix A for more details on the Cache implementation.

Part B: Multi-level Cache (2 points)

Reference execution time: 12-13 sec per trace

You will implement an instruction cache (ICACHE) and a data cache (DCACHE), both of which are instances of part A's Cache. You will connect the two L1 caches to a unified L2 cache, and the L2 cache to DRAM. You will also estimate timing for each request in this part. The L2 Cache implements a **Least Frequently Used** replacement policy. In case multiple cache lines are tied with the same lowest access frequency, the MRU cache line among them should be the victim. We will refer to this replacement policy as "LFU+MRU".

Your objective is to write two functions in memsys.cpp:

1. **memsys_access_modeBC**, which returns the delay required to service a given request.
2. **memsys_L2_access,** which is called by memsys_access_modeBC and returns the delay for servicing a request that accesses the L2 cache.

Note that for the purpose of calculating delay, you can assume that writebacks are completed off the critical path and hence do not account for the accumulation of delay for a read request.

NOTE: All caches are Write-back and Allocate-On-Miss. For this part assume a fixed DRAM Latency of 100 cycles.

Part C: Implementing a Simple DRAM (3 points)

Reference execution time: 12-13 sec per trace

For Part C, you will model a simple row buffer for the DRAM memory system. You will assume that the DRAM memory has 16 banks and the memory system could follow either open page or close page policy. Address mapping to DRAM follows a cache block interleaving policy: i.e., consecutive cache blocks map to consecutive banks.

Your objective is to implement the DRAM class in dram.{cpp, h}. **memsys_access_modeCDE** returns the DRAM delay for **both open page and close page policy. This can be controlled by a command line argument**. By default, it should return the fixed value (100 cycles) as mentioned in part B.

Refer to Appendix B for more details on the DRAM implementation.

Part D: Making the system Multicore (2 points)

Reference execution time: 26-27 sec per trace mix

You will assess the effectiveness of your memory system for a multicore processor. In particular, you will implement a two-core processor and evaluate your design for three mix workloads: Mix1 (libq-bzip2), Mix2 (libq-lbm), and Mix3 (bzip2-lbm). You will model simple LFU+MRU replacement in the shared L2 cache (as in part B) and report the performance of each core/workload pair. Pay attention to the memory traffic and the memory row buffer hit rate for the mix workload compared to the isolated workloads from Part C.

Note: There is no need to implement a coherence protocol. We use a pseudo-translation layer (memsys_convert_vpn_to_pfn()) that produces physical addresses unique to each core. Since there is no physical address overlap between cores, there is no need to maintain coherence information.

Your objective is to write two functions in memsys.cpp:
1. **memsys_access_modeDE**, which returns the delay required to service a given request.
2. **memsys_L2_access_multicore,** which is called by memsys_access_modeDE and returns the delay for servicing a request that accesses the L2 cache.

Part E: Implement Static Way Partitioning (2 points Extra Credit)

Reference execution time: 26-27 sec per trace mix

Shared caches can cause an ill-behaved application to completely consume the capacity of the shared cache and cause significant performance degradation to the neighboring application. This can be mitigated with way partitioning, whereby each core can be given a fixed quota of ways per set. In this part you will implement static way partitioning for a system consisting of two cores. We will provide “SWP_core0ways” as the quota for core 0 (the remaining N-SWP_core0ways becomes the quota of core 1).

Note 1: Invalid lines are still the prime candidates for selection, even if selecting an invalid line as victim results in a core’s quota being temporarily exceeded.
Note 2: When selecting a victim among multiple valid lines within a partition, use LFU+MRU as your replacement policy.

Your objective is to update **cache_find_victim** in cache.cpp to handle SWP.

How to run the simulator:

1. ./sim -h (to see the simulator options)
2. ./sim -mode 1 ../traces/*.mtr.gz (to test the default configuration)
3. ../scripts/runall.sh runs all configurations for all traces
4. ../scripts/runall.sh takes nearly 30 minutes to execute.

WHAT TO SUBMIT FOR Parts A+B+C:

For phase 1 (parts A, B, C) you will submit a compressed folder:

1. src_ABC.tar.gz, containing the src folder with all source code files and the makefile:
src_ABC.tar.gz:
    src/*.cpp
    src/*.h
    src/makefile

WHAT TO SUBMIT FOR Part D (+E):

For phase 2 (parts D, E) you will submit a compressed folder:
1. src_DE.tar.gz, containing the src folder with all source code files and the makefile:
src_DEF.tar.gz:
    src/*.cpp
    src/*.h
    src/makefile

REFERENCE MACHINE:

Gradescope autograder:
A simple autograder on Gradescope generates a score based on the difference of the reference outputs and your code’s outputs for a subset of all traces. Please note that the autograder is not comprehensive and your code will be tested against additional traces.

You can use the virtual machine **oortcloud.cc.gatech.edu** to develop and test your code. Your submission must compile, run, and produce the correct results on Gradescope. Please ensure that your submission on Gradescope produces the desired output (without any extra printf statements).

Hints:

1. We have provided last_access_time as a means for tracking the time, which can be used to implement the LRU (and MRU) replacement policy.

2. We need to track the number of times the cache line has been accessed to implement LFU. Assume that each cache line has a 6-bit saturating counter (first access to a cache line sets the counter to 0, every next access increments it until the value 2^6-1 is reached).

2. You will need the last_evicted line for Part B, when you must schedule a writeback from the Dcache to the L2cache, and from the L2 cache to DRAM.

Appendix A: Cache Model implementation

In the “src” directory, you need to update two files:

- cache.h
- cache.cpp

The following data structures may be needed for completing the Lab. You are free to use any name, any function (with any number of arguments) to implement a cache. You are free to deviate from these structural definitions as well.

1. “**Cache Line**” structure (Cache_Line), will have the following fields:
   - **Valid**: denotes if the cache line is indeed present in the Cache
   - **Dirty**: denotes if the latest data value is present only in the local Cache
   - **Tag**: denotes the conventional higher-order address bits beyond Index
   - **Core ID**: needed to identify the core to which a cache line (way) is assigned to in a multicore scenario (required for Part D, E)
   - **Last_access_time**: to keep track of when each line was inserted, which helps with the LRU/MRU replacement policy
   - **Frequency:** to track the number of accesses to the specific cache line, which helps with the LFU replacement policy
2. “**Cache Set**” structure (Cache_Set), will have:
   - **Cache_Line Struct** (replicated “# of Ways” times, as in an array/list)
3. The overarching “**Cache**” structure should have:
   - Cache_Set Struct (replicated “#Sets” times, as in a list/array)
   - # of Ways
   - Replacement Policy
   - # of Sets
   - Last evicted Line (Cache_Line type) to be passed on to next cache hierarchy level for an install if necessary

Status Variables (Mandatory variables required for generating the desired final reports as necessary. Aimed at complementing your understanding of the underlying concepts):

- **stat_read_access**: Number of read (lookup accesses do not count as READ accesses) accesses made to the cache
- **stat_write_access**: Number of write accesses made to the cache
- **stat_read_miss**: Number of READ requests that lead to a MISS at the respective cache
- **stat_write_miss**: Number of WRITE requests that lead to a MISS at the respective cache
- **stat_dirty_evicts**: Count of requests to evict DIRTY lines

Take a look at "types.h" to choose appropriate datatypes.

NOTE: Don't change the print order/format/names of the cache_print_stats function.

Maximum supported # of WAYS should be 16.

The variables from sim.cpp might be useful:
SWP_CORE0_WAYS, cycle (You can use this as timestamp for LRU/MRU).
You can access them using 'extern'.

Appendix B: DRAM model

In the "src" directory, you need to update two files:

- dram.h
- dram.cpp

The following data structures may be needed for completing the Lab. You are free to use any name, any function (with any number of arguments) to implement the DRAM. You are free to deviate from these structural definitions as well.

1. "**Row Buffer**" Entry structure (Rowbuf_Entry) can have following entries:
   - Valid
   - Row ID (If the entry is valid, which row)

2. "**DRAM**" structure can have the following fields:
   - Array of Rowbuf Entry (Maximum could be 256)

Status Variables (Mandatory metric variables required to be implemented and updated as necessary):

- **stat_read_access**: Number of read (lookup accesses do not count as READ accesses) accesses made to the DRAM
- **stat_write_access**: Number of write accesses made to the DRAM
- **stat_read_delay**: keeps track of cumulative DRAM read latency for subsequent incoming READ requests to DRAM (only the latency spent at DRAM module)
- **stat_write_delay**: keeps track of cumulative DRAM write latency for subsequent incoming WRITE requests to DRAM (only the latency paid at DRAM module)

NOTE: Don't change the print order/format/names of the dram_print_stats function.

DRAM latencies for this part:

ACT (RAS)	**45**
CAS	**45**
PRE	**45**
BUS	**10**

Note: A single DRAM access incurs the indicated BUS latency only once (i.e., the value represents a roundtrip latency - do not add the bus latency twice, assuming you pay the latency once to get to the DRAM and once for data to return from DRAM).

Row Buffer Size = 1024
DRAM Banks = 16

The following variables from sim.cpp might be useful: SIM_MODE, CACHE_LINESIZE, DRAM_PAGE_POLICY. You can access them using 'extern'.