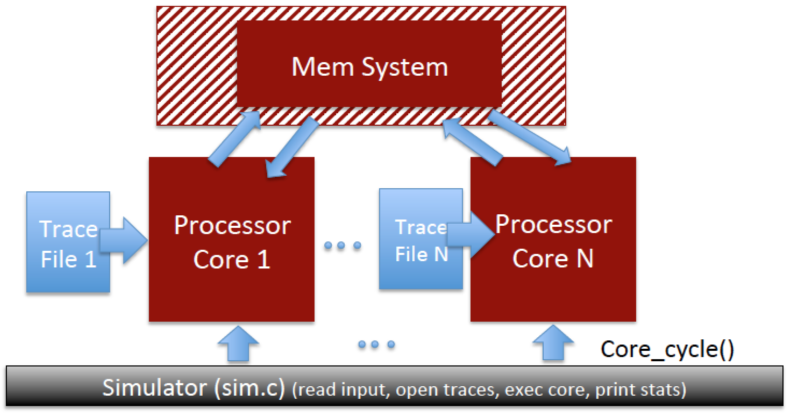
ECE 4100 / ECE 6100 / CS 4290 / CS 6290
Advanced Computer Architecture

Lab 4: CMP Memory System Design (10 (+ 5) Pts)

Parts A + B + C Due: Tuesday, November 22, 2022 (11:59 pm ET)
Part D (+ E + F) Due: Thursday, December 1, 2022 (11:59 pm ET)

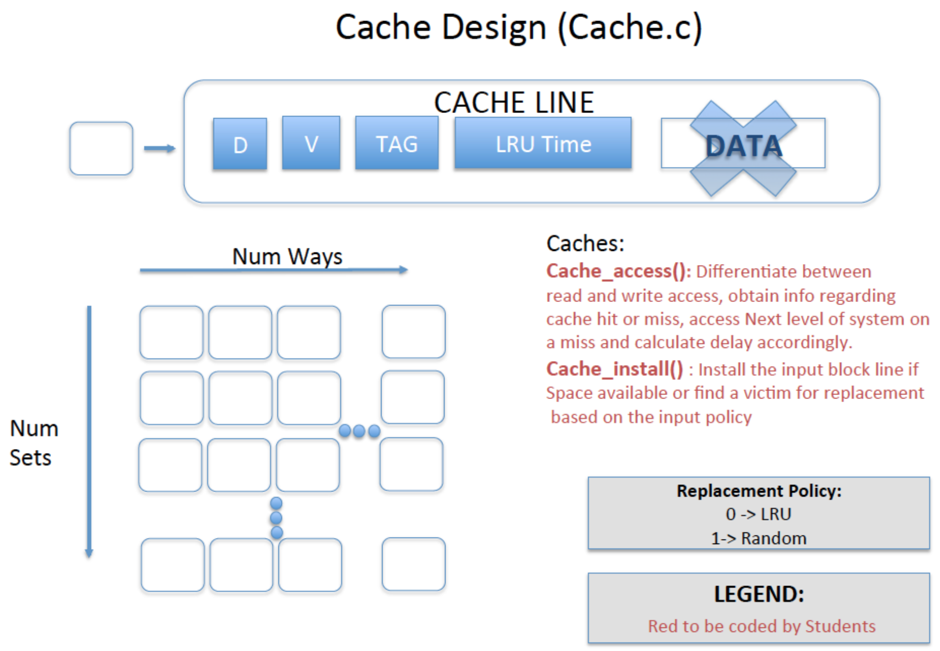
As part of this lab assignment, you will build a multi-level cache simulator with DRAM based main memory. The system will then be extended to incorporate multiple cores, where each core has a private L1 (I and D) cache, and a shared L2 cache. Misses and writebacks from the shared L2 cache are serviced by a DRAM based main memory consisting of **16 banks** and per-bank row buffers.



We will build this simulator in two phases. First Phase (A, B, and C) is for a single core system and the Second Phase (D, E, and F) extends the system to consist of multiple cores.

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

In this part, you will build a cache model and estimate the cache miss rate. You need to implement your own data structure for the cache (named **Cache**) in cache.{cpp, h} and leave all other files untouched (with the potential exception of runall.sh). The cache functions/methods must be written such that they will work for different associativity, line size, cache size, and for different replacement policies (such as LRU and Random).
Look at Appendix A for more details on the cache implementation.


Cache Design (Cache.c)
CACHE LINE
D
V
TAG
LRU Time
DATA
Num Ways
Num Sets
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 we will not be storing data values in our cache. We will provide you with traces for 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 will provide you with the trace reader and the ability to change cache parameters from command line. The tracer calls the memory system for LD 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**. At 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 job is to implement three functions in cache.cpp

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

Part B: Multi-level Cache (2 points)

You will implement an ICACHE, DCACHE, and connect it to a unified L2 cache and DRAM. You will also estimate timing for each request in this part.

Your job 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 done 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-Write-Miss
For this part assume Fixed DRAM Latency = 100

Part C: Implementing Simple DRAM (3 points)

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 policy or close page policy.

Your job is to implement DRAM class in dram.{cpp, h}
memsys_access_modeCDEF which 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 take the fixed value as mentioned in the previous part.

Look at Appendix B for more details on DRAM implementation.

Part D: Making the system Multicore (2 points)

You will conduct 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 a simple LRU based replacement in the shared L2 cache and report the weighted speedup under LRU replacement. 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.

Your job is to write **memsys_access_modeDEF** in memsys.cpp, which returns the delay required to service a given request.

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

Shared caches can cause a badly behaving application to completely consume the capacity of the shared cache, causing significant performance degradation to the neighboring application. This can be mitigated with way partitioning, whereby each core can be given a certain number of ways per set as their quota. 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).

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

Part F: Dynamic Way Partitioning (3 points Extra Credit)

You need to implement any partitioning scheme different from Static Way Partitioning. One option is to implement a Utility-Based Cache Partitioning Scheme. You can read the paper, understand the scheme, and implement it whatever way you wish to. You are free to implement anything else by looking through other technical papers or even try something of your own.

You are allowed to add any structure you would like to implement the scheme. The only restrictions would be that you are not allowed to change the size, associativity, number of sets, or line size for any of the caches. All you have to do is to implement your choice of a non-static partitioning scheme which would decide how many ways of the L2 cache are dedicated to which core. *Also note that no code debugging help would be provided for this part.*

You will implement this scheme ensuring the code behaves as expected for the other modes of the Lab as well. Your scheme should be invoked by setting the command line parameters "-mode 4 -L2repl 3", i.e., as presented to you in the runall.sh file.

Write a report briefly describing the technique you have implemented, references, your idea on why it would provide better performance and the L2 miss percentages observed by running your code on the given traces.

The grading for this part **will depend on your report**, your understanding of the new scheme and **your implementation**.
+3 points if your implementation is correct and if you get a miss percentage lower than that obtained by Parts D and Part E (for SWP_CORE0WAYS = 50%) by more than 1% for any one of the three instruction mixes.

+1 point if your implementation is correct but you do not beat Parts D and E for any instruction mix.

How to get the simulator running

1. Download the tarball "Lab_4.tar.gz"
2. `tar -xvzf Lab_4.tar.gz`
3. `cd Lab_4`
4. `cd src`
5. `make`
6. `./sim -h` (to see the simulator options)

7. `./sim -mode 1 ../traces/libq.mtr.gz` (to test a configuration)
8. `../scripts/runtests.sh` runs all test cases with reference results
9. `../scripts/runall.sh` runs all configurations for all traces and generates results

WHAT TO SUBMIT FOR Parts A+B+C:

For Phase 1 (parts A, B, C) you will submit 2 folders:

1. src_ABC.tar.gz with updated cache.cpp, memsys.cpp and dram.cpp
   Use the following steps:
   1. `make clean`
   2. `cd ..`
   3. `tar -cvzf src_ABC.tar.gz src`
2. results_ABC.tar.gz (which is a tarball of your results for the three parts, for each of the three trace files; 12 .res files in total)
   Use the following steps:
   1. `../scripts/runall.sh`
   2. `cd ..`
   3. `tar -cvzf results_ABC.tar.gz results/A.*.res results/B.*.res results/C.*.res`

WHAT TO SUBMIT FOR Part D (+E + F)

For Phase 2 (parts D, E, F) you will submit 2 folders:

1. src_DEF.tar.gz with updated cache.cpp, memsys.cpp and dram.cpp
   Use the following steps:
   1. `make clean`
   2. `cd ..`
   3. `tar -cvzf src_ABC.tar.gz src`
2. results_DEF.tar.gz (which is a tarball of your results for the three parts, for each of the three trace files; up to 15 .res files in total)
   Use the following steps:
   1. `../scripts/runall.sh`

        2. `cd ..`
        3. `tar -cvzf results_ABC.tar.gz results/D.*.res results/E.*.res results/F.*.res`
    3. DWP_report.txt (for Part F)

FAQ:

1. You can use the timestamp method to implement LRU replacement. We have provided current_cycle as a means for tracking the time.
2. You will need the last_evicted line for Part B, when you will need to schedule a writeback from the Dcache to the L2cache, and from the L2cache to DRAM.
3. For RANDOM replacement, you may get a minor change in the miss rate compared to what is shown in the report.
4. You must follow the submission file names to receive full credit. However, you will not be penalized if Canvas automatically adds a numeric suffix to the filenames.

Appendix A: Cache Model implementation

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

- cache.h
- cache.cpp

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 (e.g., CacheLine) will have 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 copy
    - **Tag**: denotes the conventional higher order PC bits beyond Index
    - **Core ID**: This shall be needed to identify the core to which a cache line (way) is assigned to in a multicore scenario (required for Part D, E, F)
    - **Last Access Time**: helps to keep track of individual last access times of the cache lines, which helps identify the LRU way
2. “**Cache Set**” structure, (e.g., CacheSet) will have:
    - **CacheLine Struct** (replicated “# of Ways” times, as in an array/list)
3. Overarching “**Cache**” structure should have:
    - CacheSet Struct (replicated “# of Sets” times, as in an array/list)
    - # of Ways
    - Replacement Policy
    - # of Sets
    - Last evicted Line (CacheLine type) to be passed on to next higher cache hierarchy 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 caches

- **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.

Following functions would be needed for caches:

Cache *cache_new(uint64_t size, uint64_t associativity, uint64_t line_size, ReplacementPolicy replacement_policy)
Allocate memory to the data structures and initialize the required fields. (You might want to use calloc() for this)

CacheResult cache_access(Cache *c, uint64_t line_addr, bool is_write, unsigned int core_id);
the system provides the cache with the line address. Returns HIT if access hits in the cache, MISS otherwise. Also if is_write is TRUE, then marks the resident line as dirty. Updates appropriate stats

void cache_install(Cache *c, uint64_t line_addr, bool is_write, unsigned int core_id);
Installs the line: determine victim using replacement policy (LRU/RAND). Copies victim into last_evicted_line for tracking writebacks
Finds victim using cache_find_victim
Initializes the evicted entry
Initializes the victim entry

unsigned int cache_find_victim (Cache *c, unsigned int set_index, unsigned int core_id);
You may find it useful to split victim selection from install

void cache_print_stats (Cache *c, const char *label);
Implemented for you

NOTE: cache_print_stats function should be implemented to print the same messages. (Don't change the print order/format/names)

Maximum # of WAYS should be 16.

The variables from sim.cpp might be useful:

SWP_CORE0_WAYS, current_cycle (You can use this as timestamp for LRU); (You can access these and any other global variables from sim.cpp using 'extern')

Appendix B: (DRAM model)

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

- dram.h
- dram.cpp

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 (e.g., RowbufEntry) can have following entries:
    - Valid
    - Row ID (If the entry is valid, which row)

2. "**DRAM**" structure can have following fields:
    - Array of Rowbuf Entry

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 paid 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)

Following functions would be needed for dram:

DRAM *dram_new();

Allocates memory to the data structures and initialize the required fields. (You might want to use calloc() for this)

uint64_t dram_access(DRAM *dram, uint64_t line_addr, bool is_dram_write);
*You may update the statistics here, and also call **dram_access_mode_CDEF** function*

uint64_t dram_access_mode_CDEF(DRAM *dram, uint64_t line_addr, bool is_dram_write);
You might need it for Part C and later
Assumes a mapping with consecutive lines in the same row
Assumes a mapping with consecutive rowbufs in consecutive rows
You may use this function to track open rows
You may compute delay based on row hit/miss/empty

void dram_print_stats(DRAM *dram);
Implemented for you

NOTE: dram_print_stats function should be implemented to print the same messages. (Don't change the print order/format/names)

DRAM Latencies for this part:

ACT	**45**
CAS	**45**
PRE	**45**
BUS	**10**

Row Buffer Size = 1024
DRAM Banks = 16

The following variables from sim.cpp might be useful: SIM_MODE, CACHE_LINESIZE, DRAM_PAGE_POLICY; (Can access these and any others using 'extern')