CS520 Computer Architecture

Project 2 – Spring 2019

Due: 5/2, 11:59 pm

1. RULES

(1) All students must work alone. Cooperation is not allowed.
(2) Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. The TAs will scan source code through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.
(3) You must do all your work in the C/C++.
(4) Your code must be compiled on remote.cs.binghamton.edu or the machines in the EB-G7 and EB-Q22. This is the platform where the TAs will compile and test your simulator. As I know, they all have the same software environment.

2. Project Description

In this project, you will implement a flexible cache and memory hierarchy simulator and use it to compare the performance, area, and energy of different memory hierarchy configurations, using a subset of the SPEC-2000/2006 benchmark suite.

3. Simulator Specification

Design a generic cache module that can be used at any level in a memory hierarchy. For example, this cache module can be "instantiated" as an L1 cache, an L2 cache, an L3 cache, and so on. Since it can be used at any level of the memory hierarchy, it will be referred to generically as CACHE throughout this specification.

3.1. Configurable parameters

CACHE should be configurable in terms of supporting any cache size, associativity, and block size, specified at the beginning of simulation:

- SIZE: Total bytes of data storage.
- ASSOC: The associativity of the cache (ASSOC = 1 is a direct-mapped cache).
- BLOCKSIZE: The number of bytes in a block.

There are a few constraints on the above parameters: 1) BLOCKSIZE is a power of two and 2) the number of sets is a power of two. *Note that ASSOC (and, therefore, SIZE) need not be a power of two*. As you know, the number of sets is determined by the following equation:

$$\#\text{sets} = \frac{SIZE}{ASSOC \times BLOCKSIZE}$$

3.2. Replacement policy

You will implement three replacement policies, LRU (least-recently-used), FIFO (first-in-first-out), and optimal replacement policy. Replacement policy will be a configurable parameter for the CACHE simulator.

A. LRU policy
Replace the block that was least recently touched (updated on hits and misses).

B. FIFO policy (Bonus: Replacement Policy)
Replace the block that was placed first in the cache.

C. Optimal policy (Bonus: Replacement Policy)
Replace the block that will be needed farthest in the future. Note that this is the most difficult replacement policy and it is impossible to implement in a real system. This will need preprocessing the trace to determine reuse distance for each memory reference (i.e. how many accesses later we will need this cache block). You can then run the actual cache simulation on the output of the preprocessing stage.

Note. If there is more than one block (in a set) that's not going to be reused again in the trace, replace the first one that comes up from the search.

3.3. Write policy

CACHE should use the WBWA (write-back + write-allocate) write policy.

- *Write-allocate*: A write that misses in CACHE will cause a block to be allocated in CACHE. Therefore, both write misses and read misses cause blocks to be allocated in CACHE.
- *Write-back*: A write updates the corresponding block in CACHE, making the block dirty. It does not update the next level in the memory hierarchy (next level of cache or memory). If a dirty block is evicted from CACHE, a writeback (i.e., a write of the entire block) will be sent to the next level in the memory hierarchy.

3.4. Allocating a block: Sending requests to next level in the memory hierarchy

Your simulator must be capable of modeling one or more instances of CACHE to form an overall memory hierarchy, as shown in Fig. 1.

CACHE receives a read or write request from whatever is above it in the memory hierarchy (either the CPU or another cache). The only situation where CACHE must interact with the next level below it (either another CACHE or main memory) is when the read or write request misses in CACHE. When the read or write request misses in CACHE, CACHE must "allocate" the requested block so that the read or write can be performed.

Thus, let us think in terms of allocating a requested block X in CACHE. The allocation of requested block X is actually a two-step process. The two steps must be performed in the following order.

S1. *Make space for the requested block X*. If there is at least one invalid block in the set, then there is already space for the requested block X and no further action is required (go to step 2). On the other hand, if all blocks in the set are valid, then a victim block V must be singled out for eviction, according to the replacement policy. If this victim block V is dirty, then a write of the victim block V must be issued to the next level of the memory hierarchy.

S2. *Bring in the requested block X*. Issue a read of the requested block X to the next level of the memory hierarchy and put the requested block X in the appropriate place in the set (as per step 1).

To summarize, when allocating a block, CACHE issues a write request (only if there is a victim block and it is dirty) followed by a read request, both to the next level of the memory hierarchy. Note that each of these two requests could themselves miss in the next level of the memory hierarchy (if the next level is another CACHE), causing a cascade of requests in subsequent levels. *Fortunately, you only need to correctly implement the two steps for an allocation locally within CACHE. If an allocation is correctly implemented locally (steps 1 and 2, above), the memory hierarchy as a whole will automatically handle cascaded requests globally.*

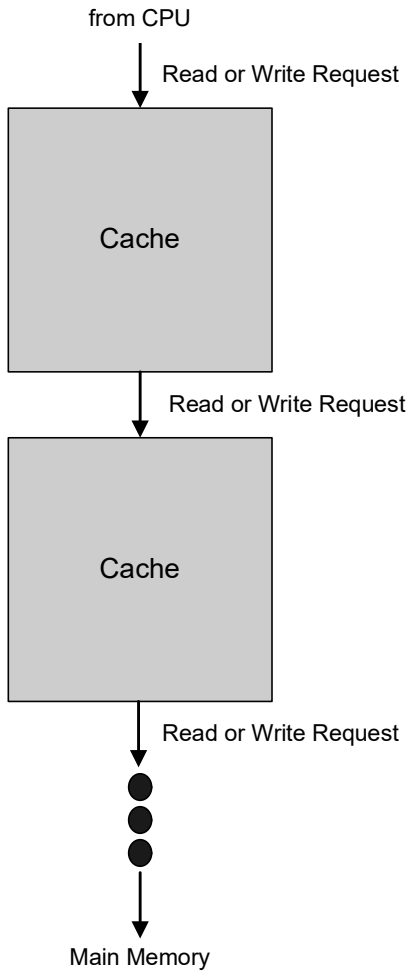

Fig 1. Your simulator must be capable of modeling one or more instances of CACHE to form an overall memory hierarchy.

3.5. Updating state

After servicing a read or write request, whether the corresponding block was in the cache already (hit) or had just been allocated (miss), remember to update other state. This state includes LRU/FIFO/optimal counters affiliated with the set as well as the valid and dirty bits affiliated with the requested block.

3.6. Inclusion property

You will implement three inclusion properties (non-inclusive, inclusive, and exclusive property) for CACHE. Inclusion property will be a configurable parameter for the CACHE simulator.

A. Non-inclusive cache

Non-inclusive property is the default property used in this project. It is simply what you'll get if you follow the directions listed above. There is no enforcement of either the cache inclusion nor the cache exclusion property. A cache block in an inner cache may or may not be in an outer cache.

B. Inclusive cache (Bonus: Inclusion property)

According to the inclusive property, an outer cache should be a superset of all inner caches it surrounds. i.e. any reference in L1 cache must also hit in the L2 cache. For homogeneous caches, such as the ones we shall be testing, the only difference between inclusive and non-inclusive cache is on L2 eviction (happens when read or write request misses at the L2 cache and the requested block needs to be allocated). When a victim block in the L2 cache needs to be evicted, the L2 cache must invalidate the corresponding block in L1 as well (assuming it exists there). If the L1 block that needs to be invalidated is dirty, a write of the block will be issued to the main memory directly.

C. Exclusive cache (Bonus: Inclusion property)

An exclusive cache is an outer cache that is guaranteed not to contain any cache blocks present in any of its inner cache(s). i.e. any reference in the L1 cache must not be present in the L2 cache, and vice versa. There are three possible scenarios:

Scenario #1: a read or write request hits in the L1 cache. In this case, the block can be read or written directly from L1.

Scenario #2: a read or write request misses in the L1 cache but hits in the L2 cache. In this case, issue a read of the requested block to the L2 cache, bring in the block to the L1 (replaced block must be placed in L2) and invalidate the block in L2. (If the block in the L2 cache is dirty, the block becomes a clean block in L1 and the dirty block in L2 is written back to the main memory when it's invalidated.)

Scenario #3: a read or write request misses both in the L1 and the L2 cache. In this case, issue a read of the requested block directly to the main memory and bring it to L1.

Note: when a clean block is evicted from the L1 cache to the L2 cache, use read request to the L2 cache, assuming we are reading a clean block from the main memory.

Now let us think about the eviction of a victim block. There are two possible scenarios:

Scenario #1: the victim block is in the L1 cache. In this case, a write of the victim block is issued to the L2 cache (which will be a miss in L2) no matter the victim block is dirty or not, and the dirty bit (either 0 or 1) will be written back with the block into L2.

Scenario #2: the victim block is in the L2 cache. In this case, a write of the victim block is issued to the main memory only if the victim block is dirty.

4. Memory Hierarchies to be Explored in this Machine Problem

While Fig. 1 illustrates an arbitrary memory hierarchy, you will only study the memory hierarchy configurations shown in Fig. 2. For this project, all CACHEs in the memory hierarchy will have the same BLOCKSIZE.

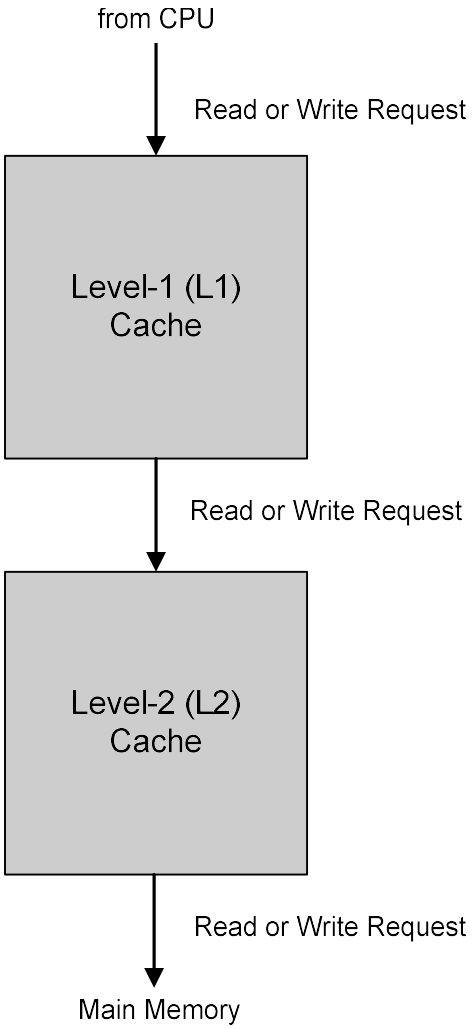

Fig 2. Configurations to be studied

5. Inputs to Simulator

The simulator reads a trace file in the following format:

r|w <hex address>
r|w <hex address>
...

"r" (read) indicates a load and "w" (write) indicates a store from the processor.
Example:
r ffe04540
r ffe04544
w 0eff2340
r ffe04548
...

NOTE:
All addresses are 32 bits. When expressed in hexadecimal format (hex), an address is 8 hex digits as shown in the example trace above. In the actual trace files, you may notice some addresses are comprised of fewer than 8 hex digits: this is because there are leading 0's which are not explicitly shown. For example, an address "ffff" is really "0000ffff", because all addresses are 32 bits, i.e., 8 nibbles.

6. Outputs from Simulator

Your simulator should output the following: (see posted validation runs for exact format)
1. Memory hierarchy configuration and trace filename.
2. The final contents of all caches.
3. The following measurements:

a. number of L1 reads
b. number of L1 read misses
c. number of L1 writes
d. number of L1 write misses
e. L1 miss rate = MR_{L1} = (L1 read misses + L1 write misses)/(L1 reads + L1 writes)
f. number of writebacks from L1 to next level
g. number of L2 reads (should match b+d: L1 read misses + L1 write misses for non-inclusive and inclusive cache)
h. number of L2 read misses if there is a L2 cache
i. number of L2 writes (should match f: number of writebacks from L1 to L2)
j. number of L2 write misses
k. L2 miss rate (ignore L2 writes) = MR_{L2} = (item h)/(item g)
l. number of writebacks from L2 to memory
m. total memory traffic = number of blocks transferred to/from memory
   (with L2, should match h+j+l for non-inclusive and exclusive cache: all L2 read misses + L2 write misses + writebacks from L2
   Note: for inclusive cache, writebacks directly from L1 to memory due to invalidation should also be counted)
   (without L2, should match b+d+f: L1 read misses + L1 write misses + writebacks from L1)

7. Validation and Other Requirements

7.1. Validation requirements

Sample simulation outputs will be provided on the website. These are called "validation runs". You must run your simulator and debug it until it matches the validation runs.
Each validation run includes:

1. Memory hierarchy configuration and trace filename.
2. The final contents of all caches.
3. All measurements described in Section 7.

Your simulator must print outputs to the console (i.e., to the screen). (Also see Section 7.2 about this requirement.)

Your output must match both numerically and in terms of formatting. You must confirm correctness of your simulator by following these two steps for each validation run:

1) Redirect the console output of your simulator to a temporary file. This can be achieved by placing "> your_output_file" after the simulator command.
2) Test whether or not your outputs match properly, by running this unix command:
diff –iw <your_output_file> <posted_output_file>

The –iw flags tell "diff" to treat upper-case and lower-case as equivalent and to ignore the amount of whitespace between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some whitespace where the validation runs have whitespace.

7.2. Compiling and running simulator

You will hand in source code and the TAs will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions.

1. You must be able to compile and run your simulator on machines in EB-G7 and EB-Q22. This is required so that the TAs can compile and run your simulator. You also can access the machine with the same environment remotely at remote.cs.binghamton.edu via SSH.

2. Along with your source code, you must provide a **Makefile** that automatically compiles the simulator. This Makefile must create a simulator named "sim_cache". The TAs should be able to type only "make" and the simulator will successfully compile. The TAs should be able to type only "make clean" to automatically remove object files and the simulator executable. An example Makefile will be posted on the web page, which you can copy and modify for your needs.

3. Your simulator must accept exactly 10 command-line arguments in the following order:

   sim_cache <BLOCKSIZE> <L1_SIZE> <L1_ASSOC> <L2_SIZE> <L2_ASSOC><REPLACEMENT_POLICY> <INCLUSION_PROPERTY> <trace_file>

- o *BLOCKSIZE*: Positive integer. Block size in bytes. (Same block size for all caches in the memory hierarchy.)
- o *L1_SIZE*: Positive integer. L1 cache size in bytes.
- o *L1_ASSOC*: Positive integer. L1 set-associativity (1 is direct-mapped).
- o *L2_SIZE*: Positive integer. L2 cache size in bytes. L2_SIZE = 0 signifies that there is no L2 cache.
- o *L2_ASSOC*: Positive integer. L2 set-associativity (1 is direct-mapped).
- o *REPLACEMENT_POLICY*: Positive integer. 0 for LRU, 1 for FIFO, 2 for Optimal
- o *INCLUSION_PROPERTY*: Positive integer. 0 for non-inclusive, 1 for inclusive, 2 for exclusive
- o *trace_file*: Character string. Full name of trace file including any extensions.

Example: 8KB 4-way set-associative L1 cache with 32B block size, 256KB 8-way set-associative L2 cache with 32B block size, LRU replacement, non-inclusive cache (default), gcc trace:

sim_cache 32 8192 4 262144 8 0 0 gcc_trace.txt

4. Your simulator must print outputs to the console (i.e., to the screen). This way, when a TA runs your simulator, he/she can simply redirect the output of your simulator to a filename of his/her choosing for validating the results.

7.3. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator efficient is also important for a couple of reasons.

First, the TAs need to test every student's simulator. Therefore, we are placing the constraint that your simulator must finish a single run in 2 minutes or less. If your simulator takes longer than 2 minutes to finish a single run, please see the TAs as they may be able to help you speed up your simulator.

Second, you will be running many experiments: many memory hierarchy configurations and multiple traces. Therefore, you will benefit from implementing a simulator that is reasonably fast.

One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the web page includes the –O3 optimization flag.

Note that, when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without –O3 and with –g. Optimization includes register allocation. Often, register-allocated variables are not displayed properly in debuggers, which is why you want to disable optimization when using a debugger. The –g flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. The debugger needs this information to recognize variable names, function names, line numbers in the source code, etc. When you are done debugging, recompile with –O3 and without –g, to get the most efficient simulator again.

8. Experiments and report

8.1. L1 cache exploration: SIZE and ASSOC

GRAPH #1 (total number of simulations: 55) (10 points)
For this experiment:

- L1 cache: SIZE is varied, ASSOC is varied, BLOCKSIZE = 32.
- L2 cache: None.
- Replacement policy: LRU
- Inclusion property: non-inclusive
- Trace: GCC trace

Plot L1 miss rate on the y-axis versus log2(SIZE) on the x-axis, for eleven different cache sizes: SIZE = 1KB, 2KB, … , 1MB, in powers-of-two. (That is, log2(SIZE) = 10, 11, …, 20.) The graph should contain five separate curves (i.e., lines connecting points), one for each of the following associativities: direct-mapped, 2-way set-associative, 4-way set-associative, 8-way set associative, and fully-associative. All points for direct-mapped caches should be connected with a line, all points for 2-way set-associative caches should be connected with a line, etc.

Discussion to include in your report:
Q1. Discuss trends in the graph. For a given associativity, how does increasing cache size affect miss rate? For a given cache size, what is the effect of increasing associativity?

Q2. Estimate the compulsory miss rate.

Q3. For each associativity, estimate the conflict miss rate.

GRAPH #2 (no additional simulations with respect to GRAPH #1) (10 points)
Same as GRAPH #1, but the y-axis should be AAT instead of L1 miss rate. Refer the cacti table file to get the cache access time. Assume that the memory access time is 28ns. If the cache access time for a specific cache configuration is not in the cacti table, skip the configuration.

Discussion to include in your report:
Q1. For a memory hierarchy with only an L1 cache and BLOCKSIZE = 32, which configuration yields the best (i.e., lowest) AAT?

8.2. Replacement policy study (Bonus Replacement Policy: 20)

GRAPH #3 (total number of simulations: 27) (5 points)
For this experiment:

- L1 cache: SIZE is varied, ASSOC = 4, BLOCKSIZE = 32.
- L2 cache: None.
- Replacement policy: varied

- Inclusion property: non-inclusive
- Trace: GCC trace

Plot AAT on the y-axis versus log2(SIZE) on the x-axis, for nine different cache sizes: SIZE = 1KB, 2KB, … , 256KB, in powers-of-two. (That is, log2(SIZE) = 10, 11, …, 18.) The graph should contain three separate curves (i.e., lines connecting points), one for each of the following replacement policies: LRU, FIFO, optimal. All points for LRU replacement policy should be connected with a line, all points for FIFO replacement policy should be connected with a line, etc. Refer the cacti table file to get the cache access time. Assume that the memory access time is 28ns. If the cache access time for a specific cache configuration is not in the cacti table, skip the configuration.

Discussion to include in your report:
Q1. Discuss trends in the graph. Which replacement policy yields the best (i.e., lowest) AAT?

8.3. Inclusion property study (Bonus Inclusion Policy: 20)

GRAPH #4 (total number of simulations: 12) (5 points)
For this experiment:

- L1 cache: SIZE = 1KB, ASSOC = 4, BLOCKSIZE = 32.
- L2 cache: SIZE = 2KB – 64KB, ASSOC = 8, BLOCKSIZE = 32.
- Replacement policy: LRU
- Inclusion property: varied
- TRACE: GCC trace

Plot AAT on the y-axis versus log2(L2 SIZE) on the x-axis, for six different L2 cache sizes: L2 SIZE = 2KB, 4KB, … , 64KB, in powers-of-two. (That is, log2(L2 SIZE) = 11, 12, …, 16.) The graph should contain three separate curves (i.e., lines connecting points), one for each of the following inclusion properties: non-inclusive and inclusive. All points for non-inclusive cache should be connected with a line, all points for inclusive cache should be connected with a line. Refer the cacti table file to get the cache access time. Assume that the memory access time is 28ns. If the cache access time for a specific cache configuration is not in the cacti table, skip the configuration.

Discussion to include in your report:
Q1. Discuss trends in the graph. Which inclusion property yields a better (i.e., lower) AAT?

9. What to submit

You must hand in a single zip file called project2.zip. Below is an example showing how to create project2.zip from a Linux machine. Suppose you have a bunch of source code files (*.cc, *.h), the Makefile, and your project report (report.doc).

zip project2 *.cc *.h Makefile report.pdf

project2.zip must contain the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.):

(a) Project report. This must be a single PDF document named report.pdf. (No Word documents please.) The report must include the following:

- A cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A cover page will be posted on the project website.
- See section 8 for the required content of the report

(b) Source code. You must include the commented source code for the simulator program itself. You may use any number of .cc/.h files, .c/.h files, etc.

(c) Makefile. See Section 7.2 for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

Note:
Zip only the files listed above, not the directory containing these.

10. Penalties

Cheating: Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a 0 for the project and other disciplinary actions.