North Carolina State University

Department of Electrical and Computer Engineering

ECE463/563: Fall 2017

Project1: Cache and Memory Hierarchy Design

Design Specifications v1

Part A Submission Deadline
September 20, 2017
11:59PM

Ground rules

1. All students must work alone. The project scope is reduced (but still substantial) for ECE 463 students, as detailed in this specification.
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 (from current and past semesters) through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.
3. A Wolfware message board is provided for posting questions, discussing and debating issues, and making clarifications. It is an essential aspect of the project and communal learning. Students must not abuse the message board, whether inadvertently or intentionally. Above all, never post actual code on the message (unless permitted by the TAs/instructor). When in doubt about posting something of a potentially sensitive nature, email the TAs and instructor to clear the doubt.
4. You must do all your work in the C/C++ or Java languages. Exceptions must be approved. The C language is fine to use, as that is what many students are trained in. Basic C++ extensions to the C language (e.g., classes instead of structs) are encouraged (but by no means required) because it enables more straightforward code reuse.
5. Use of the Grendel environment is required. This is the platform where the TA will compile and test your simulator. Please test your simulator on Grendel machines before submission.

CAUTION: If you develop your simulator on another platform, get it working on that platform, and then try to port it over to Grendel at the last minute, you may encounter major problems. Porting is not as quick and easy as you think unless you are an excellent programmer. Worse, malicious bugs can be hidden until you port the code to a different platform, which is an unpleasant surprise close to the deadline. So, keep this in mind.

Contents

Project Overview

In this project, you will implement a flexible cache and memory hierarchy simulator and use it to study the performance of memory hierarchies using the SPEC benchmarks.

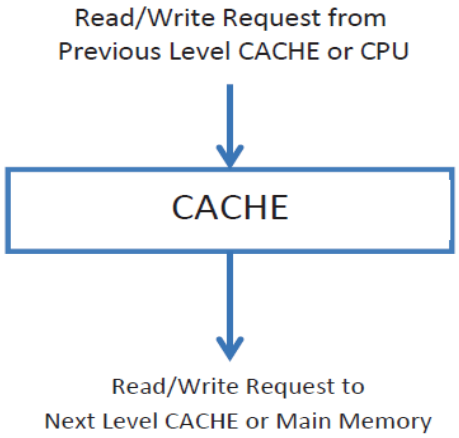
This project is divided into two parts. Both Part A and Part B are to be submitted separately. (Specifications of Part B will be released soon.) In Part A, you will design a generic cache simulator module with some configurable parameters. This cache module can be instantiated (used) as an L1 cache, an L2 cache, or 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. In Part B, you will design a flexible two-level memory hierarchy simulator with certain extensions using the CACHE module designed in Part- A.

Both simulators will take an input in a standard format which describes the read/write requests from the processor. Simulator output is also expected to be in a standard format as explained in further sections.

1. Part A: The generic CACHE module

Design a generic CACHE module that can be used at any level in a memory hierarchy. This generic CACHE can be configured using different design parameters. It takes read/write requests as input and optionally generates appropriate read/write request for the next level of memory hierarchy.

In Part A, you will design a one level cache memory hierarchy. Hence, all the read/write requests come from the CPU and the next level of memory hierarchy is always the main memory.



1.1. 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 assumptions for the above parameters:

- BLOCKSIZE is a power of two.
- The number of sets in a cache is also a power of two.
- ASSOC and SIZE need not be a power of two.

As you know, the number of sets is determined by the following equation.

$$\text{Number of Sets} = \frac{\text{SIZE}}{ASSOC \times BLOCKSIZE}$$

1.2. Configurable Policies

Apart from the configurable parameters, the CACHE can be configured in terms of policies. Corresponding policies will be specified at the beginning of the simulation.

1.2.1 Replacement Policy

All students (ECE463 and ECE563) need to implement the LRU (Least Recently Used) replacement policy. ECE 521 students will need to implement one additional policy called LFU (Least Frequently Used) replacement policy. Replacement policy will be a configurable parameter for the CACHE simulator.

1.2.1.1 LRU policy

All students (ECE463 and ECE563) must implement LRU replacement policy as discussed in class.

1.2.1.2 LFU policy

The students of ECE563 will have to implement the LFU replacement policy. The LFU replacement policy decides which block in a set is to be evicted by choosing the block that has been referenced least frequently. A per-block counter is used to keep track of the number of references (read/write) to each block in a set. This method, however, suffers from a serious problem: Blocks with high reference counts tend to stay in the cache set even after the block is not referenced for a long time, thus effectively denying a more useful block cache space.

In this project, you will implement a variation of the LFU replacement policy that overcomes the above issue, called 'LFU with Dynamic Aging.' In addition to a per-block reference counter (COUNT_BLOCK), a per-set age counter (COUNT_SET) is used to initialize the block reference counter when a block is brought into the cache.

The mechanism is as follows:

Initially, all counters are initialized to 0.

1. When a block is brought into the cache set, its reference count COUNT_BLOCK is initialized to COUNT_SET+1.
2. When a block is referenced, its reference count COUNT_BLOCK is incremented by 1.
3. Deciding which block to evict involves selecting the block having the lowest reference count COUNT_BLOCK in that set. In case of the same reference count, select the first block in the set to match TA’s output.

4. When evicting a block from a set, the set's age counter COUNT_SET is set to the reference count of the evicted block.

There are two noteworthy points:

a. The value of COUNT_SET is either equal or slightly lesser than the lowest value of COUNT_BLOCK in that set. Thus, COUNT_SET is an approximate measure of "age" of the blocks in that set. Also,COUNT_BLOCK is not just the number of references to the block, but a measure of the number of references as well as "age" of the block. Hence the name, "Dynamic Aging".
b. This mechanism solves the problem of unpopular blocks with high reference counts staying the cache indefinitely. Even if a block has a very high reference count, if it is not referenced for a long time, the COUNT_BLOCK values of other blocks in the set slowly rise (due to their initialization to COUNT_SET+1 and further references to the blocks) so that the reference count of the unpopular block eventually becomes the lowest in that set.

Source: Dilley et. al., "Enhancement and Validation of Squid Cache Replacement Policy" HP Laboratories, Palo Alto, CA, 1999

1.2.2 Write Policy

All students (ECE463 and ECE563) need to implement two write policies for the CACHE. CACHE should support the WBWA (write-back + write-allocate) and WTNA (write through + write-notallocate) write policies.

1.2.2.1 Write-Back Write –Allocate (WBWA)

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 CACHE or the main memory) at that time. If a dirty block is evicted from CACHE, a “Writeback” is performed and the entire block will be sent to the next level in the memory hierarchy.A write that misses in CACHE will cause a new block to be allocated in CACHE. Therefore, both write misses and read misses cause blocks to be allocated in CACHE.

1.2.2.2 Write-Through write Not-Allocate (WTNA)

A write will update the corresponding block in CACHE and it will also update the respective block in next level of memory hierarchy (next level CACHE or the main memory) at that time. This will never cause any block to be dirty in the CACHE. Also, if a write is missed in the CACHE, it won't cause a new block to be allocated in the CACHE. It will directly propagate to the next level.

1.3. Modeling the CACHE

This simulator can model various instances of CACHE in a memory hierarchy. For Part A, the simulator will model only a single level memory hierarchy. CACHE receives a read or write request from the higher level (CPU). Only situation where

CACHE must interact with the next level below it (main memory) is when the read or write request misses in the CACHE. CACHE always allocates a new block of data when a read request is missed. But a write-miss may or may not cause a new block to be allocated in the CACHE. This depends on the write policy.

Allocation of a new block

Think of one of the above scenarios in which CACHE needs to allocate a new block X. The allocation of requested block X is actually a two-step process. The two steps must be performed in the following order.

1. *Make space for the requested block X*. If there is at least one invalid (free) block in the set, then there is already space for the requested block X and no further action is required. (Go to step 2). To be consistent with the TA's simulation, place the requested block X in place of the first invalid block if there are more than one invalid blocks. 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 (Section 1.2). For WBWA policy, if this victim block V is dirty, then a write-back of the victim block V must be issued to the next level of the memory hierarchy.

2. *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 (determined in step 1).

1.4. Input to the Simulator

The simulator reads a trace file which follows following format.

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

Here `r` indicates a "load" and `w` indicates a "store" from the processor. The simulator must parse the trace file and issue the corresponding read/write request to the highest level in the memory hierarchy.

Example trace file

```
R ffe04540
r ffe04544
w 0eff2340
r ffe04548
r ffe04544
w 0eff2340
r ffe04548
...
```

1.5. Raw Measurements

This simulator aims at collecting the data to calculate certain statistics for the CACHE. The simulator should be able to compute following raw statistics at the end of simulation for a given configuration of CACHE. In Part A, L1 is the only CACHE in the single level memory hierarchy.

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} =(b + d)/(a + c)
f. number of write-backs from L1 to memory
g. total memory traffic = number of blocks transferred to/from memory
   Note: g should match (b + d + f) if L1 uses WBWA policy.

The simulator will print out the CACHE configuration, the statistics and the status of CACHE at the end of the simulation to output console in a specified format.

1.6. Performance Analysis

From the simulation results, you can analyze the performance of various single level cache memories.

Average Access Time (AAT)

It is the average time it takes for a CACHE to service a single read/write request from the processor. For memory hierarchy with only L1 cache, AAT can be computed using following equations.

$$\text{Total access time} = (\text{Reads}_{L1} + Writes_{L1}) \times \text{HT}_{L1} + (ReadMisses_{L1} + WriteMisses_{L1}) \times MissPenalty_{L1}$$

$$\text{L1 Miss Rate, MR}_{L1} = \frac{ReadMisses_{L1} + WriteMisses_{L1}}{Reads_{L1} + Writes_{L1}}$$

$$\text{Average Access Time} = \frac{Total\ Access\ Time}{Reads_{L1} + Writes_{L1}}$$

$$\text{Average Access Time} = \text{HT}_{L1} + (MR_{L1} \times MissPenalty_{L1})$$

Note: HT_{L1}(L1 hit-time) and the L1 Miss Penalty can be obtained from the course website for this project.

1.7. Validation Requirements

Your simulator code will be tested electronically. Trace files will be used as simulator inputs. Trace files are explained in section 1.4. Various configurations of CACHE consisting of cache size, associativity, block size, replacement policy and write policy will be tested on the trace files. Sample outputs from the simulator called “Validation Runs” will be posted on the course website.
Each validation run includes

1. The memory hierarchy configuration
2. The final contents of all caches in the memory hierarchy
3. All measurements described in Section 1.5

1.7.1 Building and Running the simulator

You will submit the source code electronically and the TA 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 (see Section 1.10 for Grading Policy).

1. You must be able to compile and run your simulator on Grendel (32 bit, x86) machines. This is required so that the TA can compile and run your simulator. If you are logging into a Grendel machine remotely and do not know whether or not it is Linux (as opposed to SunOS), use the **uname** command to determine the operating system.
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 TA should be able to build the simulator using single **make** command. The TA should be able to remove object and executable files using **make clean** command. To make your life easy, an example Makefile will be posted on the course website, which you can copy and modify according to your needs.
3. Your simulator must accept exactly 6 command-line arguments in the following order:

   sim_cache <L1_BLOCKSIZE> <L1_SIZE> <L1_ASSOC>
   <L1_REPLACEMENT_POLICY> <L1_WRITE_POLICY> <trace_file>

<L1_BLOCKSIZE>	Block size in bytes. Positive Integer, Power of two
<L1_SIZE>	Total CACHE size in bytes. Positive Integer
<L1_ASSOC>	Associativity of Cache.
<L1_REPLACEMENT_POLICY>	0 for LRU; 1 for LFU
<L1_WRITE_POLICY>	0 for WBWA; 1 for WTNA
<trace_file>	Character string. Full name of the trace file including any extensions

Example: 8KB 4-way set-associative L1 cache with 32B block size, LRU replacement policy and WTNA write policy will be simulated for “gcc_trace” with following command.

```
$ ./sim_cache 32 8192 4 0 1 ./trace/gcc_trace
```

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

1.7.2 Validation

Your output must match both numerically and in terms of formatting when compared to the validation runs. The TA will literally **diff** your output with the correct output. 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 using ">" operator. For example,

   ```
   $ ./sim_cache 32 8192 4 0 1 ./trace/gcc_trace.txt>my_output
   ```

2. Test whether or not your outputs match properly, by running this Linux command. This command must output "nothing" indicating a correct match.

   ```
   $ diff –iw my_output validation_run
   ```

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

For guidelines on how to get started with designing your simulator, see Appendix.

1.8. Simulation Run time

Correctness of your simulator is of paramount importance. That said, making your simulator efficient is also important for a couple of reasons. First, the TA needs 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 TA as they may be able to help you speed up your simulator.

Second, you will be running many experiments. 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 the code with –O3 and without –g to get the most efficient simulator again.

1.9. What to submit via Wolfware

You must hand in a single zip file called **project1A.zip** that is no more than 1MB in size. Notify the TA beforehand if you have special space requirements. However, a zip file of 1MB should be sufficient for any code.

Your **project1A.zip** must contain only the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.):

1. Source code. You must include the commented source code for the simulator program itself.You may use any number of .c/cpp/.cc/.h files.
2. Makefile. See Section 1.7, for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

Below is an example showing how to create project1.zip from a Grendel machine. Suppose you have a bunch of source code files (*.cc, *.h), the Makefile, and your project report (report.doc).

```
$ tar -zcvf project1A *.cc *.h Makefile
```

As TA will unzip and grade your code electronically using scripts your zip file must have all files inside directly (not in a folder inside the zip). So, keep this in mind if you create zip from GUI program in windows.

1.10. Grading Policy

Definition of "project works":

√ **ALL raw measurements match (Section 1.5)**

√ **The final contents of the cache match**

There will be no credit for a run if it does not work according to the definition above.

Item		Points(ECE463)	Points(ECE563)
Substantial Simulator can be compiled and run		10	10
L1 works with LRU and WBWA	Validation Run #1	7	3
	Mystery Run A	3	2
L1 works with LRU and WTNA	Validation Run #2	7	3
	Validation Run #3	3	2
L1 works with LFU and WBWA	Validation Run #4	NA	5
L1 works with LFU and WTNA	Validation Run #5	NA	3
	Mystery Run B		2
Total		**30**	**30**

Various deductions (out of 30 points):

-1 point for each day late, according to Wolfware timestamp.

Reminder: since it's Part A, whose major purpose is to split a big project into 2 parts and push you to reach a safe milestone by the date, we don't want to be harsh on late policy. In future project, the late policy is likely -1 point for each hour. Please keep in mind.

TIP: Submit whatever you have completed by the deadline just to be safe. If, after the deadline, you want to continue working on the project to get more features working and/or finish experiments, you may submit a second version of your project late. If you choose to do this, the TA will grade both the ontime version and the late version (*late penalty applied to the late version*), and take the maximum score of the two. Hopefully you will complete everything by the deadline, however.

Up to -5 points for not complying with specific procedures. Follow all procedures very carefully to avoid penalties. Complete the **SUBMISSION CHECKLIST** to make sure you have met all requirements.

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.

Appendix

Design Guidelines

Here are some guidelines to get you started with design of your CACHE simulator. In this simulator you need to simulate only the **tags** for each cache block, and you don't need to simulate any kind of data-transfer to and from CACHE. You can use C/C++ or JAVA languages for your code. In this guide, use of C/C++ languages is assumed.

The guidelines provided here are just for your reference. It's not mandatory to follow these guidelines in your project. As long as your simulator output matches with the validation runs, you are good.

In your program, CACHE can be represented as a data structure implemented with a C **struct** or a C++ **class**. Use of C++ is not necessary but is recommended as it simplifies the development process. As we are designing a generic CACHE which can be used at any level in memory hierarchy, the implementation of CACHE should be independent of the position in memory hierarchy.

For this, the CACHE data structure needs to be a node in a linked list. Each CACHE will have access to its immediate next level in memory hierarchy. This can be implemented as a pointer to the CACHE data structure inside CACHE. For CACHE just above the main memory, this pointer will simply be NULL.

In this way, your simulator will only give read/write requests to the first level CACHE (just below the CPU) using its member functions. This CACHE will forward the requests to next level if needed using the **nextLevel** pointer.

CACHE will keep track of number of reads, writes, misses, write-backs, memory accesses etc. using some counter variables. These counter variables will be incremented during the simulation as needed.

In a simulator run, it will first initialize the CACHE data structures using command line arguments. Then it will start reading the trace file and issue the read/write requests to the first level (L1 CACHE) and L1 CACHE will increment its counters and updates its tags and if needed, it will issue read/write requests to L2 CACHE (if it exists). Similarly L2 CACHE will increment its counters and updates its tags and so on. At the end of simulation when all requests from the trace file are completed, simulator will read the counter values and display raw measurements and contents of each CACHE.

```
class CACHE
{
private:
/* AddCACHE data members
add variables for parameters like
size, block size, associativity,
write and replacement policies
dynamic Array for tag storage,
all counter variables
and other variables needed
*/

//pointer to the next CACHE in memory hierarchy
CACHE *nextLevel;
public:
//CACHE member functions
bool readFromAddress(unsigned int add);
bool writeToAddress(unsigned int add);
//functions to add more
//functionality in your CACHE
};
```