North Carolina State University

Department of Electrical and Computer Engineering

ECE463/563: Fall 2017

Project1: Cache and Memory Hierarchy Design

Design Specifications v1

Part B Submission Deadline

October 1, 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.

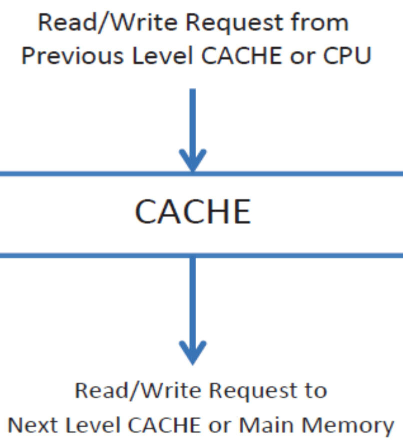
This cache module can be instantiated (used) as an L1 cache, an L2 cache, or an L3 cache, and so on. Since it can be used at any level of the memory hierarchy, it will be referred to generically as CACHE throughout this specification. In Part B, you will design a flexible two-level memory hierarchy simulator 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 563 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 less 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.

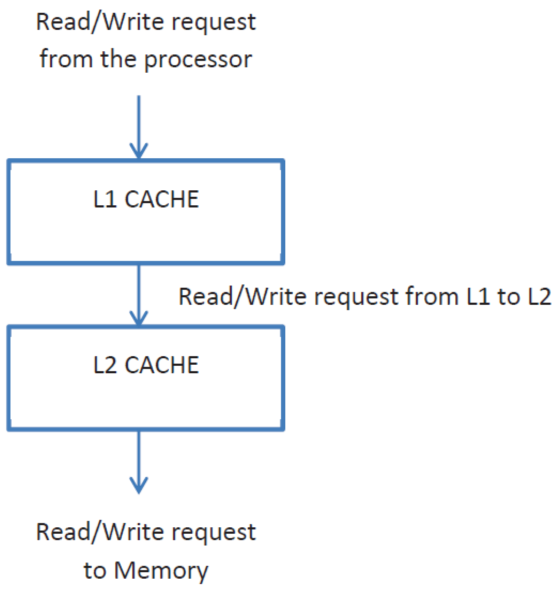
Allocation of a new block

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).

2. Part B: Two Level Memory Hierarchy



2.1. Modeling the memory hierarchy

In this part, the simulator will create two instances of the CACHE designed in Part A. The simulator will read the trace file and assign the requests to L1 cache as before. Now, L1 cache will send read/write requests to L2 cache. L2 cache will interact with the memory.

L1 and L2 cache will keep track of their own counters for reads, writes, hits, misses etc. At the end of the simulation, program will print final state of both the caches and raw statistics for both caches.

Both L1 and L2 can be configured in terms of block size (however for this project both L1 and L2 will always have the same blocksize), cache size, associativity but not replacement policy or write policy etc. These parameters will be provided from command-line.

2.2. 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 will be a power of two.

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

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

2.3. Replacement Policy

All students (ECE463 and ECE563) need to implement the LRU (Least Recently Used) as discussed in class.

Students in ECE563 are expected to implement the LFU(Least Frequently Used) replacement policy. And LRFU (Least Recently/Frequently Used) replacement policy is also encouraged to be implemented which combines both the properties of LRU and LFU. Here is a paper for you to reference: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=970573, especially in Section 3 and Section 4.

The LRFU policy associates a value with each block, and we call it CRF(Combined Recency and Frequency) value and quantifies the likelihood that the block will be referenced in the near future. The CRF value for any block B at time $\mathrm{t_{base}}$, denoted as

$$\mathrm{C}_{\mathrm{t}_{base}}(B) = \sum_{i=1}^{k} F(t_{base} - t_{B_i}),$$

where $\mathrm{F(x)}$ is a weighing function and $\{\mathrm{t_{B_1}}, \mathrm{t_{B_2}}, \dots, \mathrm{t_{B_k}}\}$ are the k reference times of block B, and $\mathrm{t_{B_1}}, < \mathrm{t_{B_2}} < \dots < \mathrm{t_{B_k}}$.

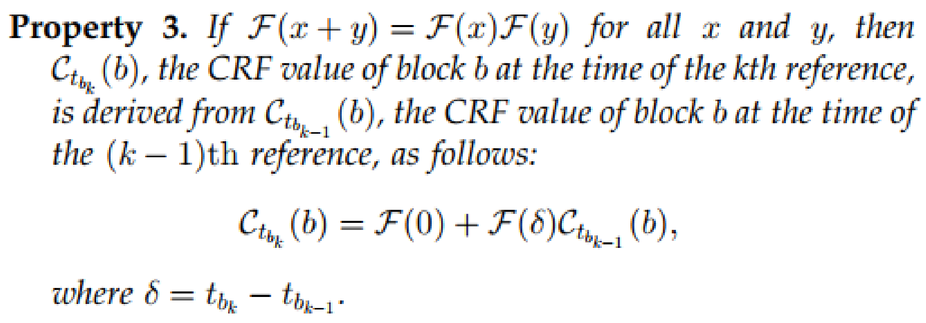
From the definition, we can see that the CRF of a block is combined with all its historical references, then in this case, all the reference will contribute to its total

CRF value, but those references in earlier times will have less impact on its overall CRF.

When we are to implement it, we noticed that we may need to record all its historical references, which is a waste of storage. Hence, we need to choose the function F(x) that satisfies F(a+b) = F(a)F(b), so that we can store the historical contributes to the CRF into one register, thus saving the storage.

In our case, we choose $\mathrm{F(x)} = \left(\frac{1}{2}\right)^{\lambda x}$ $(\lambda > 0)$, which satisfies F(a+b) = F(a)F(b). An interesting observation is that when $\lambda = 0$, $\mathrm{F(x)} = 1$, then it's degraded to LFU; when $\lambda = 1$, $\forall \mathrm{i}, \mathrm{F(i)} > \sum_{j=i+1}^{k} F(j)$, for any k where $\mathrm{k} \geq \mathrm{i} + 1$, then it's degraded to LRU. In the other aspect, when λ is approaching 0, LFU weighs more; when λ is approaching 1, LRU weighs more. Details could be seen in the paper.

Also referring from the Property 3 in the paper, we can get this conclusion:


Property 3. *If $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ for all x and y, then $\mathcal{C}_{t_{b_k}}(b)$, the CRF value of block b at the time of the kth reference, is derived from $\mathcal{C}_{t_{b_{k-1}}}(b)$, the CRF value of block b at the time of the $(k-1)$th reference, as follows:*

$$\mathcal{C}_{t_{b_k}}(b) = \mathcal{F}(0) + \mathcal{F}(\delta)\mathcal{C}_{t_{b_{k-1}}}(b),$$

where $\delta = t_{b_k} - t_{b_{k-1}}$.

Thus, In our Project, we will implement in this way:

0. Define a GLOBAL_COUNTER which counts the number of addresses referenced by the memory hierarchy(or in other words, loaded from the test file); Initiate a LAST_REF_TIMESTAMP as 0 to each block, and also initiate a CRF (dataType: double. <u>For the sake of grading, don't use float here</u>) as 0 to each block.
1. When a block is referenced at GLOBAL_COUNTER=k, we update:
    a. The CRF counter will be calculated

    $\mathrm{CRF} = 1 + \mathrm{F(k - LAST\_REF\_TIMESTAMP)} \times \mathrm{CRF};$

    b. Update $\mathrm{LAST\_REF\_TIMESTAMP} = \mathrm{k};$
2. Deciding which block to evict involves selecting the block with the lowest reference count of CRF in that set: you need to calculate and compare for each block of the set its current temporary CRF by assuming as if it were hit at GLOBAL_COUNTER=k. Then you will evict the block with the lowest temporary CRF. In case of the same reference

count, select the first block in the set to match TA's output. Then, to fetch the missed block from the lower level cache/main memory, we will update(or initiate) both the CRF and LAST_REF_TIMESTAMP of the newly brought block:

$$\mathrm{CRF} = 1; \mathrm{LAST\_REF\_TIMESTAMP} = \mathrm{k};$$

While for other blocks in this set, we restore their CRF and LAST_REF_TIMESTAMP to their previous values.

In this implementation, we choose $\mathrm{F(x)} = \left(\frac{1}{2}\right)^{\lambda \mathrm{x}}$ $(0 \leq \lambda \leq 1)$. λ will be specified in the parameter list when running the program.

By the way, currently you are only required to implement this LRFU to L1 cache(VC cache not included). Reason: for L2 cache, it's very likely that two contiguous reference timestamps for some certain block might have a large difference, leading to arithmetic issues when calculating F(x). To fix this problem it may make the project too complicated. Then in the report, you might want to compare the L1-cache performance of LRFU with LRU/LFU with different value of λ chosen.

2.4. Write Policy

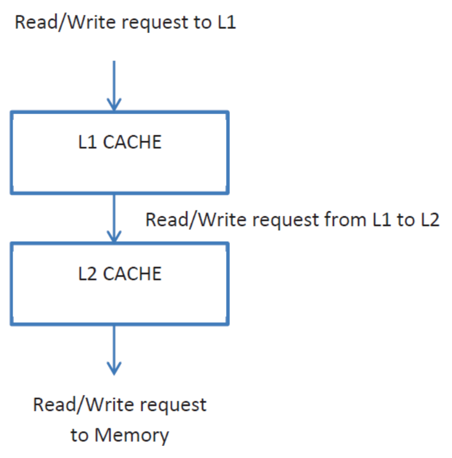
All students (ECE463 and ECE563) need to implement WBWA (write-back + write-allocate) write policy.

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.

2.5. 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 figure below


Read/Write request to L1
L1 CACHE
Read/Write request from L1 to L2
L2 CACHE
Read/Write request to Memory

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.

1. 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 (Section 2.3). 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.
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 (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.

3. ECE563 Students: Augment L1 Cache with Victim Cache

Student enrolled in ECE563 must additionally augment L1 cache with a victim cache

3.1 Victim cache can be enabled or disabled

Your simulator should be able to specify whether or not its victim cache is enabled.

3.2 Victim cache parameters

The victim cache is a fully associative cache. It uses the same block size as L1 cache. Setting its size=0 disables the victim cache.

3.3 Operations on victim cache

A victim cache is a small fully associative cache. It will uses LRU replacement policy, which means that you don't need to model LFU replacement policy for this victim cache. However, it has some specific operations. There are following possible scenarios:

Scenario #1: Request hits in L1 cache: In this case, nothing happens to victim cache.

Scenario #2: Request misses in L1 cache but hits in victim cache: In this case, swap the hit block in victim cache and LRU block in L1. Because victim cache also uses LRU replacement policy, the counter should be updated properly. There is no need to access next level memory.

Example: the current L1 cache and victim cache look like as following

A(3) (Dirty)	B(0) (Dirty)	C(1)	C(2)

L1 Cache

E(2)	F(0)	G(1)	H(3)

Victim Cache

The new request is read Address E. This request misses in L1 but hits in victim cache. Because block A is LRU block in L1, we need to swap the block A in L1 and block E in victim cache. Then they should look like this

E(0)	B(1) (Dirty)	C(2)	C(3)

L1 Cache

A(0) (Dirty)	F(1)	G(2)	H(3)

Victim Cache

Scenario #3: Request misses in L1 cache and misses in victim cache: In this case, we access the next level memory hierarchy to get the desired cache block. But before accessing next level memory hierarchy, we should also consider the following subscenarios.

Scenario #3.1: L1 cache does not evict any cache block. This scenario could happen at the very beginning when there are invalid cache blocks in L1. In this case, we just need to replace one invalid cache block in L1 based on replacement policy. There is no need to evict any cache block in L1. Read the desired cache block from next level memory hierarchy directly.

Scenario #3.2: L1 cache does evict cache block. In this case, the evicted block from L1 goes to victim cache. In order to make room for evicted block from L1, the victim cache needs to evict its LRU block. If the LRU block in victim cache is dirty, write this dirty block back to next level hierarchy at first. Then read the desired block from next level memory hierarchy.

Example: the current L1 cache and victim cache look like

A(3)	B(0) (Dirty)	C(1)	D(2)

L1 Cache

E(2)	F(0)	G(1)	H(3) (Dirty)

Victim Cache

The new request is read Address K. This request misses in L1 and misses in victim cache. L1 evicts block A and block A goes to victim cache. The victim cache should evict block H to make room for block A. Because block H is dirty, we need to write block H back to next level hierarchy. Then they should look like this

K(0)	B(1) (Dirty)	C(2)	D(3)

L1 Cache

E(3)	F(1)	G(2)	A(0)

Victim Cache

4. Configurations to be test

ECE 463: L1 cache + L2 cache (LRU+WBWA)

ECE 563: L1 cache + L2 cache

L1 cache + victim cache

L1 cache + victim cache + L2 cache

(*)L1 cache + victim cache + L2 cache + LRFU Policy for L1 Cache

Note: For ECE563 students, Both L1 and L2 Caches should be implemented with LRU and LFU policies, both using WBWA. Victim Cache is fixed with LRU+WBWA. Also both L1 and L2 will be of the same replacement policy, meaning that if LRU(or LFU) is chosen, both L1 and L2 caches will use LRU(or LFU), EXCEPT LRFU, as it is only used for L1. When LRFU is used in L1, L2 will use LRU. The LRFU is a bonus one that you are encouraged to but not required.

ECE 463 students are encouraged to challenge the requirements of ECE 563 students, but not mandatory.

5. Validation Requirements

`<L1_BLOCKSIZE>`	Block size in bytes. Positive Integer, Power of two
`<L1_SIZE>`	Total CACHE size in bytes. Positive Integer
`<L1_ASSOC>`	Associativity of the cache.
`<Victim_Cache_SIZE>`	Total victim cache size in bytes.0 signifies victim cache is disabled
`<L2_SIZE>`	Total CACHE size in bytes.0 signifies L2 is disabled
`<L2_ASSOC>`	Associativity of the cache. 0 represents L2 disabled
`<λ>`	λ parameter($0 \leq \lambda \leq 1$) for LRFU. If $\lambda = 2$: LRU; If $\lambda = 3$: LFU.
`<trace_file>`	Character string. Full name of the trace file including any extensions

Example:

```
$./sim_cache 64 1024 2 128 4096 8 0.5 ./trace/gcc_trace.txt
```

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 file for validating the results.

Validation

Same requirement as Part A also applies for Part B.

6. Printing Instructions

While printing the contents of cache please make sure that you print the cache block tags in the order of their LRU counts. Make sure that you print the most recently used block first (therefore the least recently used block at the last). i.e. For any particular set print the block tag which has LRU = 0 (most recently used) first and then print blocks having LRU=1 and so on.

For ECE 563 Students:

Follow the same instruction for printing the victim cache contents.

Note: Project 1 Part A cache contents were not printed in a similar fashion so please make sure that you make the appropriate changes to your print function so that your runs match exactly with TA's validation runs.

7. Raw measurements

For Part B you need to gather following statistics from the simulation.

a. number of L1 reads
b. number of L1 read misses, excluding L1 read misses that hit in victim cache if victim cache is enabled
c. number of L1 writes
d. number of L1 write misses, excluding L1 write misses that hit in victim cache if victim cache is enabled

e. L1 miss rate = MRL1 = (L1 read misses + L1 write misses)/(L1 reads + L1 writes)
f. number of swap (requests miss in L1 but hit in victim cache)
g. number of writeback from victim cache to next level memory
h. number of L2 read
i. number of L2 read miss
j. number of L2 write
k. number of L2 write miss
l. L2 miss rate (from standpoint of stalling the CPU) = MRL2 =(item i) / (item h)
m. number of l2 writeback to main memory
n. total memory access

(with L2, should match: L2 read misses + L2 write misses + writeback from L2)

(without L2, should match: L1 read misses + L1 write misses + writeback from victim cache)

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

8. Validation Requirements

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

8.1 Average Access Time (AAT)

It's the average time it takes for the memory hierarchy to service a single read/write request from the processor. For memory hierarchy with only L1 and L2 cache, AAT can be computed using following equations.

Average Access Time = HTL1 + (MRL1 *(HTL2+MRL2*Miss PenaltyL2))

Note: HTL1 , HTL2 and Miss PenaltyL2 can be obtained from the course website for this project.

8.2 Simulation Run time

Same requirements as Part A applies.

9. Experiments and Report

You need to submit a combined report for part A and part B. You will perform experiments with your simulator. You will present, analyze, and discuss the results in a written report.

You need to perform following experiments for each benchmark trace. (gcc_trace, perl_trace, go_trace, vortex_trace)

9.1.1 Explore the effect of following parameters on overall performance of cache

Using your simulator, explore the performance of different memory hierarchy design for the following parameters:

For ECE 463 and ECE 563 students1

1. L1 Cache size vs. miss rate (For different associativities) [Without L2]
2. Associativity vs. miss rate
3. L2 Cache size vs. miss rate (keep L1 size constant)

For ECE 563 students

1. Vary size of victim cache vs miss rate (Keep L1 ,L2 constant)

9.1.2 Thoroughly explore the design space and discuss noteworthy trends

Using your simulator, explore the memory hierarchy design space and collect result (raw measurements and AAT) for each configuration that you evaluate. Only evaluate configurations that fit within the Area Budget posted on the project website (Area ≤Area Budget). In your written report, include following.

a. Include results for all the configurations that you evaluate, using tables and graphs. Please present results in a way that reveals noteworthy trends as different parameters are varied. Graphs are best for displaying trends.

b. Discuss noteworthy trends in terms of the influence of different parameters on raw measurements and AAT.

9.1.3 Find best memory hierarchy configuration

From your thorough design space exploration, find the Best Memory Hierarchy for AAT (for Area $\leqslant$ Area Budget) for each trace. In your written report, include following.

a. Highlight the configuration that gives the lowest AAT.
b. Closely examine the configuration, its raw measurements, etc., and from this closer examination explain why it gives the lowest AAT compared to all other configurations.

What are unique aspects of this configuration that enable it to achieve the lowest AAT?

9.1.4 Compare and contrast different benchmarks

Are the Best Memory Hierarchies very similar or very different for different benchmarks? Can you conclude anything about the different benchmarks based on their Best Memory Hierarchies (e.g., is anything revealed regarding the degree of compulsory misses, capacity misses, or conflict misses)?

9.2 What to submit via Wolfware

You must hand in a single zip file called project1B.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 project1B.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.
3. Project Report. Should contain the data, graphs and analysis.

Below is an example showing how to create project1B.zip from an Eos Linux machine. Suppose you have a bunch of source code files (*.cc, *.h), the Makefile, and your project report(report.doc).

```
$ tar  -zcvf  project1B *.cc *.h Makefile report.doc
```

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.

9.3. 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		20	20
L1, L2 works with no victim cache (LRU and LFU)	Validation Run #6	10	4
	Validation Run #7	10	4
	Mystery Run C	10	2
L1 works with Victim Cache and no L2 (LRU and LFU)	Validation Run #8	+1	3
	Mystery Run D		2
L1, L2 works with victim cache (LRU/LFU)	Validation Run #9	+3 in total. No partial.	6
	Validation Run #10		6
	Mystery Run E		3
L1, L2 works with victim cache, LRFU enabled	Validation Run #11	+1 in total No partial.	+3 in total No partial
	Validation Run #12		

with different parameters	Mystery Run F		
Project Report		20	20
Total		**70 + 5**	**70 + 3**

Various deductions (out of 70 points):

-1 point for every 2 hours late, according to Wolfware timestamp.

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 -10 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.

```cpp
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
};
```