Project - Cache Organization, Code Execution and Result Analysis

In this assignment, you will become familiar with how caches work and how to evaluate their performance. To achieve these goals, you will first build a cache simulator and validate its correctness. Then you will use your cache simulator to study many different cache organizations and management policies as discussed in lecture and in Chapter 5 of Hennessy & Patterson.

Section 1 will walk you through how to build the cache simulator, section 2 will specify the protocol in which your program should connect with the user and section 3 will give you instructions on making sense out of your simulation's results.

Note that a "cache line" is equivalent to a cache entry which could either be already inside the cache or waiting to be placed inside of it.

1 Cache Simulator

In the first part of this assignment, you will build a cache simulator. The type of simulator you will build is known as a *trace-driven* simulator because it takes as input a trace of events, in this case memory references. The trace, which we will provide for you, was acquired on another machine. Once acquired, it can be used to drive simulation studies. In this assignment, the memory reference events specified in the trace(s) we will give you will be used by your simulator to drive the movement of data in and out of the cache, thus simulating its behavior. Trace-driven simulators are very effective for studying caches.

Be aware that since evaluation of your simulator's performance will be done by a computer judge, your cache simulator should be configurable based on arguments given as input, must give its results in the same exact way as the sample outputs and needs to support the following functionalities:

- Total cache size
- Block size
- Unified vs. split I- and D-caches (Von Neumann. vs. Harvard)
- Associativity
- Write back vs. write through
- Write allocate vs. write no-allocate

In addition to implementing the functionalities listed above, your simulator must also collect and report several statistics that will be used to verify the correctness of the simulator, and that will be used for performance evaluation later in the assignment. In particular, your simulator must track:

- Number of instruction references
- Number of data references
- Number of instruction misses
- Number of data misses
- Number of words fetched from memory
- Number of words copied back to memory

1.1 Files

There are three trace files that you could use to test your simulator. Their names are "spice.trace," "cc.trace," and "tex.trace." These files are the result of tracing the memory reference behavior of the spice circuit simulator, a C compiler, and the TeX text formatting program, respectively. They represent roughly 1 million memory references each. For your testing purposes, you could slice any part of the files and use them as test cases, but be aware that testing the entirety of any one of these files could take a very long time and would probably use a significant portion of your computer's resources, so use them carefully.

The trace files are in ASCII format, so they are in human-readable form. Each line in the trace file represents a single memory reference and contains two numbers: a reference type, which is a number between 0–2, and a memory address. All other text following these two numbers should be ignored by your simulator. The reference number specifies what type of memory reference is being performed with the following encoding:

0	Data load reference
1	Data store reference
2	Instruction load reference

The number following the reference type is the byte address of the memory reference itself. This number is in hexadecimal format, and specifies a 32-bit byte address in the range 0-0xffffffff, inclusive.

1.2 Building the Cache Model

There are many ways to construct the cache model. You will be graded only on the correctness of the model, so you are completely free to implement the cache model in any fashion you choose. In this section, we give some hints for the implementation.

1.2.1 Incremental Approach

The most important hint is a general software engineering rule of thumb: **build the simulator by incrementally adding functionality**. A big mistake you can make is to try to implement the cache functions all at once. Instead, build the very simplest cache model possible, and test it thoroughly before proceeding. Then, add a small piece of functionality, and then test that thoroughly before proceeding. And so on until you've finished the assignment. We recommend the following incremental approach:

1. Build a unified, fixed block size, direct-mapped cache with a write-back write policy and a write-allocate allocation policy.
2. Add variable block size functionality.
3. Add variable associativity functionality.
4. Add split organization functionality.
5. Add write through write policy functionality.
6. Add write no-allocate allocation policy functionality.

You can test your cache model at each stage by comparing the results you get from your simulator with the validation numbers which we will provide.

1.2.2 Adding Associativity and LRU Replacement

Once you have built a direct-mapped cache, you can extend it to handle set-associative caches by allowing multiple cache lines to reside in each set. One way to accomplish this task it to use doubly linked-lists in the place of the set. Your simulator, however, should never allow the number of cache lines in each linked list to exceed the degree of associativity configured in the cache.

If you need to insert a cache line into a set that is already at full capacity, then it is necessary to evict one of the cache lines. In the case of a direct-mapped cache, the eviction is easy since there is at most one cache line in every set. When a cache has higher associativity, it becomes necessary to choose a cache line for replacement. In this assignment, you will implement an *LRU replacement policy*. One way to implement LRU is to keep the linked list in each set sorted by the order in which the cache lines were referenced. This can be done by removing a cache line from the linked list each time you reference it, and inserting it back into the linked list at the head. In this case, you should always evict the cache line at the tail of the list.

One final hint: if you implement a set associative cache whose associativity can be configured, then you have also implemented a fully associative cache. A fully associative cache is simply an *N*-way set-associative cache with 1 set, and in which *N* is the total number of cache lines in the cache.

2 Code Execution

You should now have a cache simulator that can be configured (for total size, block size, unified versus split, associativity, write-through versus write-back, and write-allocate versus write-noallocate). Furthermore, you should verify your simulator against the sample cache results that we have given you before submitting your codes.

For more information on write policies and write-miss policies please visit:
https://en.wikipedia.org/wiki/Cache_(computing)#Writing_policies

2.1 Input Format

Your code should read inputs in the following form, note that entries marked with <>s should be replaced with user-defined inputs while the dashes are constant and should be entered to separate the inputs from each other.

First, your program should read a line as shown below:

<block size> **-** <unified or separated> **-** <associativity> **-** <write policy> **-** <write_miss policy>

Explanation for each of the entries:

<block size>: Cache's block size in bytes which should be a power of 2. (e.g. 8, 64, 1024...)

<unified or separated>: An indicator of cache's architecture. "0" Denotes Von Neumann and "1" denotes Harvard.

<associativity>: An integer indicating how associative the cache is. Remember that the cache's size (explained shortly) should be divisible by its associativity.

<write policy>: A string that should be either "wb" (write-back) or "wt" (write-through).

<write_miss policy>: A string that should be either "wa" (write-allocate) or "nw" (no-write allocate)

Then, if <unified or separated> is 0, your program should read a line as follows:

<unified size>

Which is the cache's size in bytes while being a power of 2.

Otherwise, if <unified or separated> is 1, your program should read the following line:

<instruction cache size> **-** <data cache size>

Both of which are the respective caches' sizes in bytes while being powers of 2.

Then, your program should assume the cache is empty and start reading trace entries as specified in Section 1.1. Some examples are provided alongside this file for the matter to be even clearer. In some of the examples, there exists a third column which is only a human-readable guidance to explicitly explain the cache's response to the requests and is of no other value.

2.2 Output Format

Your program should print its result of the cache's statistics using standard output (e.g. printf in C, System.out.Print in Java, etc.). Note that we assume a 32-bit architecture, therefore a block is made out of 4 bytes - as used in the TRAFFIC part of the output samples that are provided alongside this file-. Also remember that the "demand fetch" and "copies back" parts of the samples refer to the number of words fetched from memory and the number of words copied back to memory respectively (as specified in Section 1).

3 Result Analysis

3.1 Working Set Characterization

In this result analysis exercise, you will characterize the working set size of the three sample traces given.

Using your cache simulator, plot the hit rate of the cache as a function of cache size. Start with a cache size of 4 bytes, and increase the size (each time by a factor of 2) until the hit rate remains insensitive to cache size. Use a split cache organization so that you can separately characterize the behavior of instruction and data references (i.e., you will have two plots per sample trace–one for instructions, and one for data). Factor out the effects of conflict misses by *always* using a fully associative cache. Also, always set the block size to 4 bytes, use a write-back cache, and use the write-allocate policy.

Answer the following questions:

1. Explain what this experiment is doing, and how it works. Also, explain the significance of the features in the hit-rate vs. cache size plots.
2. What is the total instruction working set size and data working set size for each of the three sample traces?

3.2 Impact of Block Size

Set your cache simulator for a split I- D-cache organization, each of size 8 K-bytes. Set the associativity to 2, use a write-back cache, and use a write-allocate policy. Plot the hit rate of the cache as a function of block size. Vary the block size between 4 bytes and 4 K-bytes, in powers of 2.
Do this for the first 10000 entries of the three traces (as specified in section 1.1.), and make a separate plot for instruction and data references.
Answer the following questions:

1. Explain why the hit rate vs. block size plot has the shape that it does. In particular, explain the relevance of *spatial locality* to the shape of this curve.
2. What is the optimal block size (consider instruction and data references separately) for each trace?
3. Is the optimal block size for instruction and data references different? What does this tell you about the nature of instruction references versus data references?

3.3 Impact of Associativity

Set your cache simulator for a split I- D-cache organization, each of size 8 K-bytes. Set the block size to 128 bytes, use a write-back cache, and use a write-allocate policy. Plot the hit rate of the cache as a function of associativity. Vary the associativity between 1 and 64, in powers of 2. Do this for the first 10000 entries of the three traces, and make a separate plot for instruction and data references.

Answer the following questions:

1. Explain why the hit rate vs. associativity plot has the shape that it does.
2. Is there a difference between the plots for instruction and data references? What does this tell you about the difference in impact of associativity on instruction versus data references?

3.4 Memory Bandwidth

Using your cache simulator, compare the total memory traffic generated by a write-through versus write-back cache. Use a split I- D-cache organization. Simulate a couple of reasonable cache sizes (such as 8 K-bytes and 16 K-bytes), reasonable block sizes (such as 64 and 128 bytes), and reasonable associativities (such as 2 and 4). For now, use a write-no-allocate policy. Run 4 or 5 different simulations total.

Answer the following questions:

1. Which cache has the smaller memory traffic, the write-through cache or the write-back cache? Why?
2. Is there any scenario under which your answer to the question above would flip? Explain.

Now use the same parameters, but fix the policy to write-back. Perform the same experiment, but this time compare the write-allocate and write-no-allocate policies.

Answer the following questions:

1. Which cache has the smaller memory traffic, the write-allocate or the write-no-allocate cache? Why?
2. Is there any scenario under which your answer to the question above would flip? Explain.

One Final Note: Due to the current global situation, you'll have the option to develop this project in groups of two, but there will be *bonus points* for people who develop it alone.