"The greatest enemy of knowledge is not ignorance, it is the illusion of knowledge."
–Stephen Hawking

Learning Objectives

1. Basic Cache Design and Implementation

Work that needs to be handed in (via SVN)

1. `cache.c`: contains your cache implementation
2. `cache.h`: header file (if you modify it)

Guidelines

- You may want to spend some time drawing out data structures before you begin coding.
- Follow good programming practices: comment your code, label variables and functions according to their purpose, use a debugger, and stick to a single coding style.
- It is very important that you follow directions. All of your code should be in the `cache.c` file.
- We will be testing your code using the `EWS Linux machines`, so we will consider what runs on those machines to be the final word on correctness.
- Our test programs will try to break your code. You are encouraged to create your own test scenarios to verify the correctness of your code. One good test is to run your cache with different cache line lengths and sizes, or with different levels of associativity.

Cache Simulator

In this MP you will be completing a program to simulate a cache for a processor. The simulator will read in a trace of memory references (32-bit address) and determine which are cache hits and which are cache misses. We have provided a partial implementation.

Helpful suggestions

1. We've provided you with two simple traces so that you can test the correct behavior of your cache to make sure the basics are working: (see: https://subversion.ews.illinois.edu/svn/sp13-cs398/_shared/)
   - "trace1" consist of 5 accesses to the same address (the first should miss, the rest should hit)
   - "trace2" consists of accesses to sequential addresses; run this trace varying the block size of the cache (the first access to a block should miss, the rest should hit)

   We recommend that you build some more of these simple test cases. For example (this is not an exhaustive list):
   - a trace consisting of alternating accesses to two addresses that would map to the same set in the cache e.g., 0x0 and 0x100000 (these should be all misses in a direct mapped cache but only the first two accesses should miss in a cache with associativity of 2 or greater).
   - a test with many accesses to the same set (e.g., 0x0, 0x10000, 0x20000, 0x30000) in different orders, so you can test that LRU is working properly.
2. We've provided you an input trace file generated from an execution of the gcc C compiler.
3. We've provided our results for the "gcc-trace" trace for a number of configurations. Use this to verify your implementation.
4. We've provided the beginning of verbose output from the "gcc-trace" trace. Use the `diff` tool for checking for differences between our output and yours.

Cache Simulator

A cache simulator is a program that is used by computer architects to estimate the performance of caches of different sizes for a given workload. A key idea of a simulator is that it isn't a cache design itself, it is instead a model. So we design a simulator slightly differently than a cache. First, to estimate hit rate, we don't need to track the data values, so our cache simulator isn't going to hold data values and our *address trace* consists of only the addresses accessed (*i.e.*, it does not include the values read or written). Second, instead we may track things like LRU in ways that are easy to implement in software rather than ways that are cheap in hardware.

Your cache should be able to handle variable total sizes, as well as variable block sizes and degrees of associativity. The cache size does not necessarily have to be a power of two. The number of sets will be a power of 2 to allow for easy indexing, but the number of lines per set (the number of ways), does not need to be a power of 2. For example, 20 cache lines, divided into 4 sets will have 5 lines each. For associative caches, you should implement a Least Recently Used replacement policy. LRU can be implemented in a variety of ways. The code that we provide includes a partial implementation of the following approach:

> Have a counter of accesses (LRU_counter). Have each block store a counter value representing the last time it was accessed (update this counter each time a block is accessed). Then, LRU replacement simply requires you to replace the block with the smallest counter value. (Do not worry if these counters wrap around).

We provide you a Makefile to compile the code.

- The "-g" flag tells the compiler to include debugging information so that you can use a debugger like gdb.
- The "-Wall" flag tells the compiler to print out all warnings that occur during compilation. You do not need to fix every warning that occurs, but they may be a good indication that there is a bug in your code.

The simulator expects a number of command like arguments, as shown:

```
$ cachesim <trace_file> <cache_size> <nways> <block_size> [verbose]
```

1. `trace_file` is a file containing the loads and stores the simulator will run.
2. `cache_size` is an integer representing the cache size in bytes.
3. `nways` is an integer representing the associativity of the cache.
4. `block_size` is an integer representing the number of bytes in each block.
5. `verbose` is an optional argument. If you include "verbose" as the fifth argument, the code will print out which accesses hit or missed line-by-line.

For example, if you wanted to run the trace file "gcc-trace" through a 64KB 2-way set associative cache with 64B blocks and have it provide verbose output you would run:

```
$ cachesim gcc-trace 65536 2 64 verbose
```

Specifics

There are the things that you need to do to complete the code:

1. complete the `init_cache` function.
2. implement the `extract_tag` and `extract_index` functions. The "cache" struct contains two fields that can be used to compute bitmak masks to logically extract these fields.
3. extend the `find_block_and_update_lru` function so that it updates the lru.
4. implement the `fill_cache_with_block` function.

Discussion Questions

Direct Mapped Caches

Here is a series of address references given as word addresses: 2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 6, and 11. Assuming a direct-mapped cache with 16 one-word blocks that is initially empty, label each reference in the list as a hit or a miss and show the final contents of the cache.

Index	Cache Contents
0	
1	
2	2
3	
4	4
5	21
6	
7	
8	
9	
10	
11	
12	
13	13
14	
15	

Set Associative Caches

A set-associative cache is a compromise between a direct-mapped cache and a fully-associative cache where each memory location maps to a set of cache locations. So, an *N-way set associative* cache has N cache locations in each set. If the address has m bits and the cache has 2^s sets with blocks of 2^o bytes, then the address splits up as follows:

Tag = m-s-o	Index = s	Block Offset = o

When a given set fills up, temporal locality tells us that the data which has not been accessed for the longest time is the least likely to be needed, so we use the replacement policy called *least recently used*, or LRU, that is, we assume the data we should replace is the least recently used data.

Problems

1. Why don't we split addresses as follows?

Index = s	Tag = m-s-o	Block Offset = o

2. The L1 data cache of the AMD Barcelona is a 64KB, 2-way set-associative cache with 64-byte blocks. The Barcelona supports 40-bit physical addresses (i.e., can address 1TB of memory).

   Compute the number of sets and the size of the tag, index, and block offset fields.

3. The L2 data cache of the Intel Core 2 Duo is a 2MB, 8-way set-associative cache with 64-byte blocks. The Core 2 Duo supports 36-bit physical addresses (i.e., can address 64GB of memory).

   Compute the number of sets and the size of the tag, index, and block offset fields.

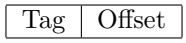
4. For the Barcelona, identify two addresses that map to the same set but are different blocks.

5. (a) **Cache 1:** Given a *direct-mapped* cache with 2 blocks of 2 bytes each, where the address is broken as follows:

Tag	Index	Offset

   If the cache was initially empty and the addresses below were accessed in that order, which of the following would be cache misses? 0110 0010 0110 1100 0111 1001 0110 1100 1101

(b) **Cache 2:** Given a fully associative cache (using a least-recently-used replacement policy) with 3 blocks of 2 bytes each, where the address is broken as shown below.



If the cache was initially empty and the addresses below were accessed in that order, which of the following would be cache misses? 0110 1101 0111 1001 1110 1100 0110 1111 1101

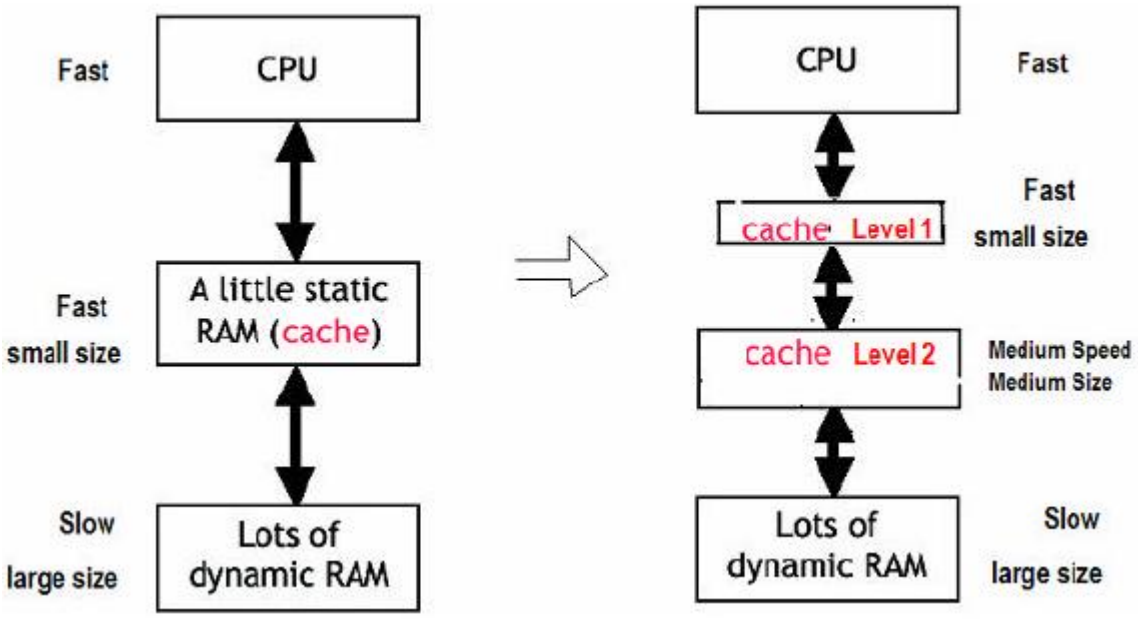
6. Give an access sequence for which Cache 1 has strictly more hits than Cache 2, or argue that such a sequence cannot exist.

Multilevel Cache

Cache is helpful only if the needed data is available there. On a cache miss, the processor still needs to access memory to retrieve the data. Each memory access is slow and results in high miss penalty. There are two ways to lower the amount of time lost on cache misses:

1. Decrease the **miss rate**. This implies building larger caches. But as the size of the cache increases, so does the average time to access it.
2. Decrease the **miss penalty**. We can't just make memory faster, but we can put another, larger layer of cache on top of the slow memory. Even a very large cache is much faster than memory. This concept, called **multilevel cache**, is illustrated below.

Figure 1. Concept of a multilevel cache



On each load and store, the processor needs to access data in memory. First, it checks Level 1 cache (L1). If the data is found, it is read or written there. On L1 cache miss, the processor accesses Level 2 cache (L2) instead of memory. Since L2 is large, it is very likely that the data can be found there. For L2 cache, read/write time is longer than L1 but much shorter than memory. Thus the overall memory access time is lower.

With this cache hierarchy, we achieve good balance between size and speed:

1. By keeping the size of L1 cache small, we retain the high speed memory access to keep the CPU busy.
2. Enlarging the overall cache size by having large L2 cache provides a good backup for L1 cache misses without slowing down the response time on L1 cache hits.

Thus we can avoid the worst case: memory access on a cache miss.

Can we add more levels of cache to achieve even better performance?
The optimum number of levels of cache is determined by the trade-off between cache size and access latency. In practice, 3 levels is pretty common in modern designs with L1=32-64KB, L2=256KB-2MB, and L3=2MB-32MB.