This lab assignment teaches how to handle large software and also refreshes your understand of the memory hierarchy in modern machines.

Part One

Setting up the Environment

The choices that we make in deciding the use of Cache affects the performance of a processor. In the first part of the project, you will modify the parameters of the cache hierarchy on X86 architecture. The work will be based on the gem5 simulator. Relevant links are

- https://www.gem5.org/
- https://research.cs.wisc.edu/multifacet/papers/can11_gem5.pdf

You will install gem5 on the VM and do the experiments on the VM. You can also do the same on your local machine and home but we will evaluate on the VM.

Since we will evaluate performance of various configurations and policies, we need to use a standard set of benchmarks.
The benchmarks is given in a separate link to be shared shortly.

There are benchmark suites also which you should use to test your instance.

- http://euler.slu.edu/~fritts/mediabench
- http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml
- http://axbench.org/https://asc.llnl.gov/CORAL-benchmarks/
- http://math.nist.gov/scimark2/

The cache design parameters you can modify are as follows:

- CPU Types: Different CPU models.
- Cache levels: No cache at all, one level or two levels.
- Size: Cache size, one of the most important choices.
- Associativity: Selection of cache associativity
    - Direct mapped, 2-way set associative, etc
- Block size: Block size of the cache, usually 64 or 32 bytes .

Evaluate CPI

For the baseline CPU configuration given below, we evaluate the Cycles per Instruction (CPI)

- CPU Models: containing TimingSimpleCPU (timing), DerivO3CPU (detailed), and MinorCPU (minor).
- Cache levels: Two levels.
- Unified caches: Separate L1 data and L1 instruction caches, unified L2 cache. (default)
- Size: 128KB L1 data cache, 128KB L1 instruction cache, 1MB unified L2 cache.

- Associativity: Two-way set-associative L1 caches, Direct-mapped L2 cache.
- Block size: 64 bytes (applied to all caches).
- Block replacement policy: Least Recent Used policy (LRU)

Use the **SE** mode for this part.

Deliverables

Given an L1 miss penalty of 6 cycles, L2 miss penalty of 50 cycles, and one cycle cache hit/instruction execution, use the configuration parameters as above and calculate the CPI for each benchmark by CPI equation.

Optimize CPI for each benchmark

By exploring different configurations (i.e. changing parameters of the cache), you will discover an optimal cache configuration for each of the 5 benchmark programs.

Deliverables

Given a 2 level hierarchy of 512 KB for L1 (both I and D combined) and 8 MB for L2:

- Identify optimal configuration to achieve lowest CPI for each benchmark
- Use GNUPlot to show tradeoffs between different design points

Part Two

In this part of the assignment you will first develop a simulation software for cache memory and then integrate the developed Replacement policy in gem5.

Simulating a Cache

The simulated cache will have the functionality that was discussed above but with a *new Replacement Policy* that works as follows. Each Cache Set is divided into two groups:

1. One group contains the HIGH PRIORITY lines of the set
2. The other group contains the LOW PRIORITY lines of the set

How is *priority* established? If a line is accessed again after the initial access that fetches it into the cache, it is promoted to the HIGH PRIORITY group. If a line is not accessed for sufficiently long (T cache accesses) after being moved to the HIGH PRIORITY group, it is moved to the LOW PRIORITY group. Within a priority group, the Least Recently Used policy may be used to manage the lines.

Make any reasonable assumptions you need. The above priority scheme is just an outline; it may not be complete. Feel free to add other details to make this work.

Inputs to the software will be:

1. Cache size

2. Cache line/block size
3. Cache associativity
4. A file containing a sequence of memory access requests: Memory address, R or W (for Read or Write), and Data (if Write)
5. The value of T

Outputs from the software will be:

1. The complete content of the cache (Data, Tag, and Control bits of every cache line) in each Way after the memory access sequence is complete.
2. Cache statistics:
    a. Number of Accesses
    b. Number of Reads, Read Hits, and Read Misses
    c. Number of Writes, Write Hits, and Write Misses
    d. Hit Ratio

Deliverables:

1. Software implementation of the cache described above
2. Document describing the design. In the document, discuss your implementation of the replacement policy. Are the High and Low priority groups fixed in size? That is, in an 8-way associative cache, are there 4 high-priority and 4 low-priority groups? Or some other break-up? Why? Can these sizes change at runtime?

Integrate your replacement Policy in Gem5

Change the default Replacement Policy (LRU) in gem5 with the code that you have developed. Run the benchmarks with the optimal cache configuration obtained in Part One and compare with CPI numbers. In case the CPI numbers differ by more than 1%, then find the lowest CPI number by iterating over the design choices to find the the optimal configuration.

Deliverables:

- CPI numbers for new scheme as a GNU Plot contrasted with the default scheme.
- In the new scheme of things, identify optimal configuration to achieve lowest CPI for each benchmark
- Use GNUPlot to show tradeoffs between different design points