6.823
Computer System Architecture
Lab 4

Assigned May 2, 2016 Worth 7% of Course Grade ***Due May 12, 2017***

http://csg.csail.mit.edu/6.823/

Warning: This lab is open-ended, and uses multi-programmed benchmarks that may take longer to complete. Do not wait until the last minute to attempt to complete the lab.

Summary

The cache hierarchy is a crucial subsystem of multicore processors. Architects face many design decisions and tradeoffs when designing the cache hierarchy, and different applications may prefer widely different hierarchies. In this lab, you will use and modify a Pin-based simulator to explore some of these issues. Specifically, you will use a simplified version of zsim (http://zsim.csail.mit.edu) to design a three-level cache hierarchy for a four-core chip with simple in-order cores.

Setting up

First, set up your environment for Pin and zsim. You'll need to do this each time you log in to work on the lab.

```
% add 6.823 && source /mit/6.823/Spring17/setup.sh
```

To obtain the materials for lab 4, use the following commands, assuming that you start in your individual repository (cd $USER) from the previous lab:

```
% svn export $LAB4ROOT
% svn add lab4handout
% svn commit -m "Lab 4 Initial Check-in"
```

In the lab4handount directory that was just created, you should find several directories, including the simulator (zsim/), some sample configurations (config/) and testing scripts (scripts/). To build the simulator, use the following command:

```
% cd lab4handout/zsim
% scons -j4
```

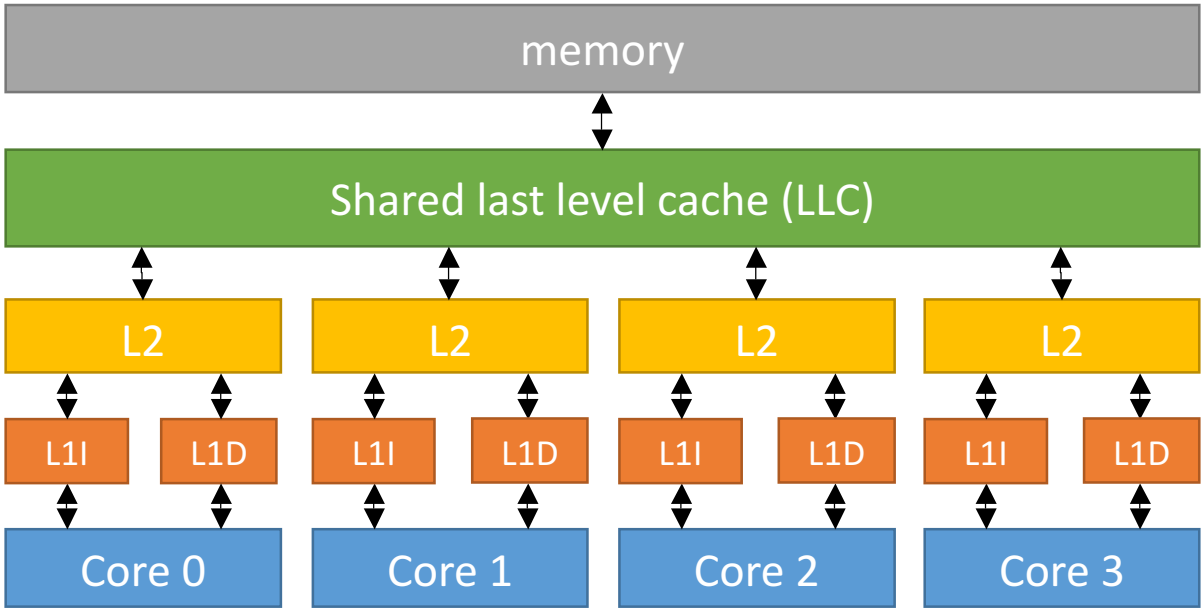
When developing your code, you may want to run zsim directly. zsim takes a single argument: a configuration file that specifies the system and programs to run. For example, you can run:

```
% cd lab4handout/config
% ../zsim/build/opt/zsim zsim.cfg
```

To make things easier, we have provided several scripts (scripts/) to generate configuration files and evaluate the simulation results, so you do not have to understand the format of config files. These scripts are described below.

Lab Task

In this lab, you will design the following three-level cache hierarchy, with L1 and L2 caches private to each core, and a last-level cache shared among all four cores:



The default code already implements inclusive caches that use the random replacement policy. By default, the L1s are 32KBs and 4-way set-associative, the L2s are 1 MB and 8-way set-associative, and the LLC is 4 MB and 16-way set-associative. All caches use 64-byte lines. Your task is to improve the performance of this hierarchy by changing the configuration of the L2s and LLC (the configuration of the L1s is fixed). To make things easier, we will test your improved cache hierarchy running different mixes of single-threaded programs that do not share memory, so *you do not need to implement a cache coherence protocol*.

Here are a few ideas that you can try to improve performance:

1) Reallocate cache capacity between L2s and LLCs. You can use larger or smaller caches, with two constraints. First, combined, all L2s and the LLC must not exceed 8MB of capacity (this is the amount of 64-byte lines they can cache; tags and replacement policy information are not included in the budget). For example, you can use 512 KB L2s with a 6 MB LLC, 256 KB L2s with a 7 MB LLC, and so on. Second, larger caches take longer to access, so you must use the latencies shown in Table 1 below.

2) Implement a better L2 and LLC replacement policy. An obvious choice is to use LRU instead of random replacement, but prior work has proposed many policies that outperform LRU, and many recent processors do not use LRU. If you change the replacement policy, you may not use more than 32 bits per cache line to store replacement information (this is a lax limit; most replacement policies work with far fewer bits per line).

3) Implement non-inclusive or exclusive caches to use limited cache space better. You can refer to the first and second section in [1] and use them as a quick review if you are not familiar with how non-inclusive or exclusive caches work.

L2 size	L2 access latency	LLC size	LLC latency
256 KB	7 cycles	7 MB	24 cycles
512 KB	11 cycles	6 MB	23 cycles
1024 KB	15 cycles	4 MB	21 cycles
2048 KB	18 cycles	2 MB	18 cycles

Table 1. Access latencies of different cache sizes.

These suggestions are enough to get you started, but we encourage you to implement and evaluate alternative techniques that improve performance further. Other optimizations are fair game, so long as they require reasonable overheads and have a reasonable implementation. If in doubt, ask the TA!

To avoid making this lab purely about exploring huge design spaces, we will limit a couple of design parameters. First, you must use 64-byte cache lines. Second, you cannot change the associativity of each cache (8-way L2s and a 16-way LLC).

Finally, a hint on approaching this lab: many of the design decisions that you face have strong interactions. For example, whether you use inclusive, non-inclusive or exclusive caches might make some replacement policies or combinations of cache sizes more desirable than others. Optimizations that you try in isolation may not work well together, and conversely, optimizations that have little effect in isolation may interact positively when used together and yields a greater improvement than in isolation. It will be more effective to think about these interactions in advance and have an implementation plan than to try a potpourri of techniques in isolation.

Evaluating Your Design

When running mixes of multiple workloads, no single metric can perfectly characterize performance improvements. For example, consider a baseline two-core system that runs a mix of two applications, A and B, one in each core. On this system, both applications complete in one time unit, as shown in Figure 1 (left). A change to the baseline system affects performance as shown in Figure 1 (right): Application A speeds up and now takes 0.67 time units, while application B slows down, taking 1.1 time units.

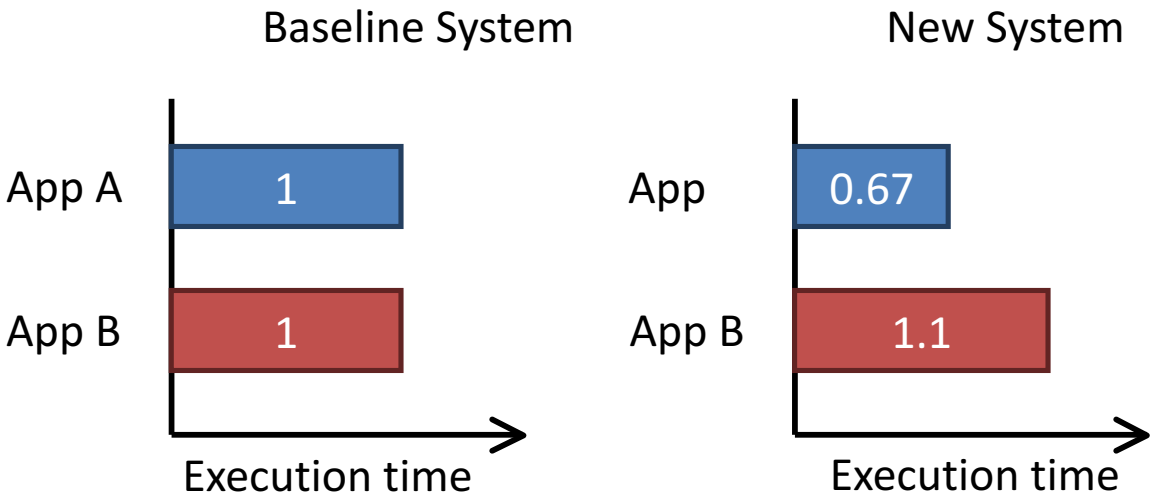


Figure 1. Example of how changes in system configuration affect the performance of different concurrent applications.

What is the overall effect on performance in this case? Application A speeds up more than application B slows down, so one could say the new system is beneficial overall. Ultimately, it depends on how the user values the performance changes of each application. To ease the evaluation of systems running workload mixes, there are a few standard ways to summarize performance [2].

In this lab, we will use **weighted speedup**, a commonly used metric to summarize performance, to evaluate your design. Weighted speedup is simply the average speedup of each application in the mix:

$$Weighted\ Speedup = \frac{1}{n}\sum_{i=1}^{n}\frac{App\ i\ exec\ time\ in\ baseline\ system}{App\ i\ exec\ time\ in\ new\ system} = \frac{1}{n}\sum_{i=1}^{n}\frac{IPC_i^{new}}{IPC_i^{baseline}}$$

In the previous example, the weighted speedup is 1.2:

$$Weighted\ Speedup = \frac{1.5 + 0.9}{2} = \mathbf{1.2}$$

Since we have 3 different multi-programmed workloads in this lab, we will take the geometric mean of the weighted speedup over the 3 workload mixes. **To get full credit, your design must have a 1.2 geometric mean weighted speedup over the baseline design.**

Hitchhiker's Guide to the Code

To complete this lab, you only need to modify the Cache class in zsim/src/cache.{h,cpp}. Each cache object receives requests from its children, and either satisfies them locally if they hit, or issues an access to its parent memory object (which can be the next-level cache or main memory) if they miss. The Cache class has the following main data members and functions:

tagArray: A 2D array storing cache tags.

parent: The parent cache/main memory this cache bank connects to.

access(): This function performs a cache access: it performs a tag lookup to see whether the access is a hit or a miss, and on a miss, invokes the replacement policy to select which line to evict and accesses the cache's parent.
chooseEvictWay() and **updatePolicy()** implement the replacement policy. You will need to change them if you decide to implement a better replacement policy.

The simulator uses a very simple timing model. The core takes a single cycle to execute each instruction, and stalls on loads, stores, and instruction fetches that miss in the L1s until the request is satisfied. The access() function returns the latency of each request. The core does not have any latency-hiding mechanisms or tries to issue multiple independent memory requests. This is not very realistic, but it makes results easy to understand.

Notice that the access function will return the latency the cache bank sees to the core. Therefore, altering the latency that cache banks return will pollute the simulation result and disqualify your submission from the competition portion of the assignment.

We have provided several scripts to help you evaluate your design (in scripts/):

runSimple.py creates the configuration file needed to simulate a single application (bzip), as well as a bash script to invoke the simulation. The first argument should be the **L2 sizes in KB**, and the second argument should be the size of **LLC in MB**. This script can be used to quickly test your design. Example usage:

```
% #Generate a configuration with 512KB L2 and 6MB LLC
% ./runSimple.py 512 6
% cd ../
% cd run-simple-[timestamp]
% bash run-simple-[L2 size]-[LLC size]-[timestamp].sh
```

runMixes.py generates the three 4-app configurations we will use for grading. You should use this to evaluate the final implementation.

print-stats.py analyzes the stats files of a finished experiment and produces human-readable text, including the IPC of each application and the misses per thousand instructions of each cache level. Example usage:

```
% ./print-stats.py [result directory]
```

get-speedup.py compares two sets of results and shows the weighed speedup of each experiment. The first argument should be the directory of the baseline implementation. The second argument should be the directory of your implementation (usually the directory runMixes.py generates). To use it:

```
% ./get-speedup.py /mit/6.823/Spring17/Lab4BaselineResult/ [result dir]
```

Although your solution will not be graded on its performance in terms of wall clock time, you should note that your Teaching Assistants are impatient people. The TA solution runs the sample testbench in about 30 minutes on the class machines (with nominal load). For grading purposes, we will allow your pin tool to run for an order of magnitude more time than ours requires (around 5 hours). After 5 hours, we will kill your submission and assign a grade based on progress to that point. Do not write horrendously inefficient code.

When you have completed the lab to your satisfaction, **specify the sizes of your memory hierarchy in memory_hierarchy.txt** and submit your changes to the svn repository. The deadline for submission is 23:59:59 EDT 12 May 2017. We'll grade whatever code you have checked in by the deadline. **No Late Submissions will be accepted**.

Lab Questions

Your response to the lab questions should be typed in lab4questions.pdf (or lab4questions.doc) in the lab4handout directory. Some questions require coding, and as such should not be put off until the last minute.

1. **Inclusion, non-inclusion, and exclusion.** Explain how they influence the cache coherence protocol design. You can assume a MSI protocol to start with for all of them and explain the difference.

2. **Design writeup.** Explain what optimizations you've done and tried for this lab. Did you observe any interesting interaction between the different techniques you tried? If you've implemented a special replacement algorithm, explain the operation of your implementation as well.

When you have answered these questions to your satisfaction, put them in a file called lab4questions.pdf (or lab4questions.doc) in your lab4handout directory, then run the following to add them and commit them.

```
% svn add lab4questions.pdf
% svn commit -m "Lab 4 Questions Check-In"
```

As with the lab code, we'll grade whatever you have checked in by the deadline.

Lab Grading

20%: Submission compiles

50%: Competitive grade based on the weighted speedup of your implementation over the baseline code with 1024K L2 and 4MB LLC. (The result directory: /mit/6.823/Spring17/Lab4BaselineResult/) We will use the same benchmarks generated by runMixes.py.

You will get full grade if your implementation achieves a weighted speedup more than **20%. If you have a weighted speedup less than 20%, your grade will be interpolated. (i.e. 50 x [your weighted speedup] / 20%)**

30%: Quality of lab question responses.

Advice on Mine Sweeping

There may be bugs in either our code or infrastructure. If you notice any `interesting' or `unexpected' behavior it could be a problem in the code or infrastructure that we provided. Report these bugs immediately to the TA, preferably in an email with the subject 6.823 Bug Report. This will help to ensure prompt fixing of any issues that may arise.

Guides for the perplexed

http://www.pintool.org/ - Pin home page
http://tig.csail.mit.edu/twiki/bin/view/TIG/UsingSubversionAtCSAIL – an SVN tutorial

[1] Sim, Jaewoong, et al. "FLEXclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion." *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 2012.

[2] Eyerman, Stijn, and Lieven Eeckhout. "System-level performance metrics for multiprogram workloads." *IEEE Micro* 28.3 (2008): 42-53.