6.5900 (6.823)
Computer System Architecture
Lab 1

Assigned Sept. 15, 2023 ***Due Sept. 29, 2023***

http:// http://csg.csail.mit.edu/6.5900/

Warning: This lab introduces a new tool that will be unfamiliar to most of you. In addition, the simulations alone for this lab may take up to a few hours to run if you are sweeping multiple parameters depending on your implementation. Do not put it off until the last minute.

Summary

The memory bottleneck is one of the paramount design concerns in modern processors. As we saw in lectures, while processor speeds have increased exponentially over the last 40 years, main memory speeds have only increased linearly since memories have been optimized for capacity rather than latency. Since memories are slow relative to the processor and memory accesses are frequent (one per loaded instruction!), a major area of architectural research is improving the perceived latency of the memory.

In order to improve the perceived latency of memory accesses, we, as architects, must first understand the nature of memory accesses. It has been the experience of architects that software memory accesses exhibit high temporal and spatial locality. Temporal locality means that the accessed memory location is likely to be accessed again in the near future. Spatial locality means that memory locations near the address of the accessed memory location are likely to be accessed in the near future.

In this lab, we will explore the concept of memory caching which is a common and highly effective mechanism for improving memory latency. A cache is a fast, but necessarily small memory that contains recently accessed areas of memory. The cache has a much lower access and update latency than the main memory; thus, if the memory locations that the process requires are present in the cache, the processor can obtain the data in a small number of cycles (as opposed to the many hundreds of cycles it would take to access the main memory), thereby improving processor performance. Since we ultimately measure the performance of the cache by its usefulness, which is the percentage of time that it is able to satisfy processor requests, statistics like *write hit percentage* and *read hit percentage* are often used to evaluate caches.

We will evaluate the behavior of various cache organizations using **Pin**, a dynamic binary instrumentation tool provided by Intel. As always, this lab is to be completed individually. You are encouraged to discuss lab concepts with fellow classmates.

Short Introduction to Pin

In order to develop architectures that run programs efficiently, architects must thoroughly understand the properties of representative programs. There are a few approaches that can be used to evaluate the behavior of a program. One is *simulation*. In this case, a software model of a processor is built, and the program executed on the model. Simulators have the advantage of being arbitrarily detailed – in theory one could build a SPICE processor simulator. Typically, architectural simulators give cycle-accurate timing estimates. The penalty for this level of detail is simulation speed: the more detailed a software simulator is, the slower it executes programs; advanced software simulators can simulate at rates of tens of KIPS (kilo instructions per second).

A second option is *program instrumentation*. Instrumentation inserts code into a program to collect information about program characteristics. Code instrumentation may be less detailed than simulation, but is often faster to implement and enjoys faster program execution. Thus, code instrumentation can be very useful in guiding architectural decisions early in the development process, before detailed simulators are available. You've almost certainly used a simple form of manual instrumentation, by inserting print statements into a program to generate a log of program activity. However, manual instrumentation is time-consuming if you want to record the characteristics of complex programs, both in terms of writing code and collecting execution results. A better choice is to use a meta language to describe the code instrumentation and develop a tool that will efficiently instrument the target program at compile or runtime.

Pin is an industrial-grade binary instrumentation tool produced by Intel and used widely in industry and academia. Pin accepts as inputs a compiled Pintool and a generic binary executable. A Pintool consists of C++ code that tells Pin where to insert code (instrumentation) and what code to insert (analysis). At runtime, executable itself is just-in-time recompiled by Pin with the analysis inserted. The code is then executed on the host machine.

A major advantage of Pin is that it can instrument programs without requiring recompilation of the executable from its original source code. Thus, even legacy binaries can be analyzed. Since Pin executes large portions of the target program natively, it can be very fast. However, this constrains the programs analyzed to the host architecture, namely x86. Yet, Pin is surprisingly versatile: new instructions can be emulated by hijacking unused x86 opcodes. Pin can be downloaded from http://www.pintool.org/ and run on any Linux or Windows platform.

In this lab and future labs (lab 2 and 4), we will be using Pin 2.14. **If you are not familiar with Pin, we highly recommend that you watch the Pin tutorial video (recorded in Spring 2021) posted on the course website. This lab will assume that you are familiar with the contents of the video.** If you are still confused about Pin concepts or want to learn more about the API, we advise you to refer to the Pin 2.14 user guide (https://software.intel.com/sites/landingpage/pintool/docs/98612/Pin/doc/html/index.html).

Setting up

We will use Git for lab submissions. If you are not familiar with Git source control, `git help` is a good command to remember. In general, we will provide skeleton code that you will edit and check in before the submission deadline. As specified in the syllabus, no late hand-ins will be accepted, so submit early and often.

Although you can develop Pintools on your laptop or any Athena computer, we will have thirteen dedicated machines we have configured for class use. These machines are:

```
vlsifarm-01.mit.edu
vlsifarm-02.mit.edu
        …
        …
vlsifarm-08.mit.edu
```

and,

```
eecs-ath-45.mit.edu
eecs-ath-46.mit.edu
        …
eecs-ath-49.mit.edu
```

These class machines use Athena passwords, but only class members may log into them. If you have trouble using a class machine, report the issue to the TA, and try logging into another machine. Additionally, the lab starter code and your individual git repositories are stored on Athena AFS. If you are unable to access the lab files, please contact your TA to obtain the AFS permissions.

Note that the `eecs-ath` machines have much newer hardware, so it is advisable that you use them over the `vlsifarm` machines if they aren't too overloaded.

Each and every time you log in to the class machine, you must set up your environment for the labs. On the lab machines, you can do this by sourcing the file `/mit/6.823/Fall23/setup.sh`. For example:

```
% ssh <athena username>@vlsifarm-01.mit.edu
% add 6.823
% source /mit/6.823/Fall23/setup.sh
```

To obtain the materials for lab 1, use the following commands:

```
% mkdir ~/6.823 && cd ~/6.823
% git clone $GITROOT
 % cd $USER
 % cp -r $LAB1FILES ./
 % git add lab1handout
 % git commit -m "Lab 1 Initial Check-in"
```

```
% git push origin master
```

`git clone $GITROOT` will clone your repository with your user name. The `cp -r $LAB1FILES ./` will create a copy of the starter code in a directory named `lab1handout`.

In the lab1handount directory that just got created, you should find a make file, some sample source code, and a test script. To build your Pintool, type the following at the command prompt:

```
% cd lab1handout
% make
```

The `caches` Pintool can be invoked from the command line in the following manner:

```
% pin.sh -injection child -ifeellucky
     -t caches -- [target_executable]
```

We have provided a test perl script lab1test.pl. The perl script will invoke Pin using the caches Pintool on multiple SPEC binaries. To invoke the perl script, type:

```
% ./lab1test.pl
```

Although we provide a script that will test your Pintool, the script will not verify your Pintool, and we will not release the expected results of the test cases (which is what would happen in industry if you were designing the next-generation microarchitecture). Further, we reserve the right to run test cases not included in the released test script. You are encouraged to compare your results with your classmates.

Lab Task

Overview

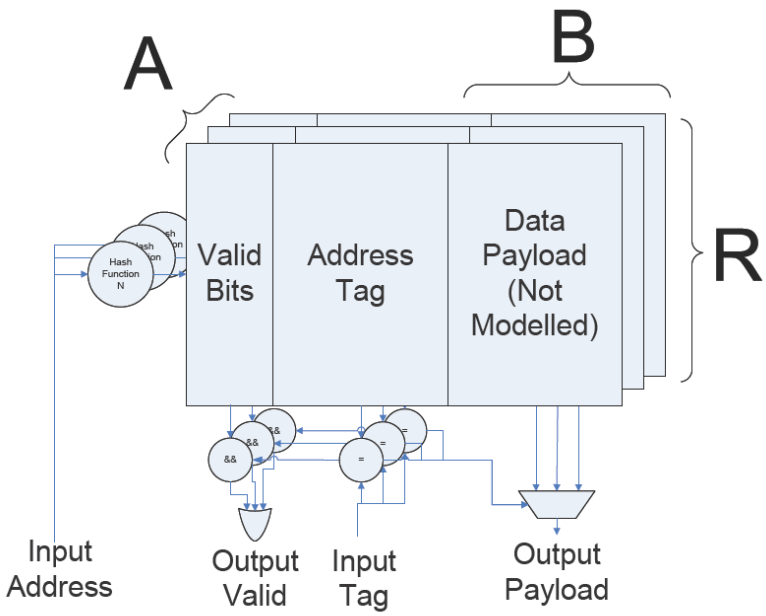
The purpose of this lab is to generate a set of parametric cache models that will help us analyze the behavior of various cache configurations. A cache can be described in terms of a relatively small number of parameters. We can view a cache as a data array containing R *sets* (or *rows*), each of which contains A (the set associativity or *ways*) data blocks of size B and some metadata (tag, valid bit) about B. When the data array is presented an address, it outputs the associated block data. The address can be broken down as <tag><index><block offset>.

The index bits are used to access a particular set of the cache. Each cache also defines a replacement policy that determines which existing block in a set to evict on a cache fill. Commonly used replacement policies include random replacement, least-recently-used, and not-most-recently-used. The following diagram depicts cache organization in terms of these parameters.

The behavior and implementation of the cache will also depend on some system-wide parameters. The caches that use virtual addresses must know the size of a memory

page in the system; depending on the hardware the size of virtually indexed cache may be constrained by this parameter (Why?). Although the size of the physical memory does not affect cache organization, it will affect cache performance as pages are swapped in and out of the physical memory.

In the lab, you will design a set of parameterized L1 data cache models based on the diagram. Note that, in the diagram, the input memory addresses accessed could be either virtual or physical. We have covered the distinction between virtual and physical addresses in class (Lectures 4-5). We will experiment with both styles of cache in this lab.



Parametric Cache Model

Starter Code

With all of the above background, we can discuss the starter code, which is located in caches.cpp. All the caches that you will implement will inherit the CacheModel base class. Notice that the base class implements some of the cache metadata for you, although you will have to add some metadata of your own. The following is a description of the CacheModel functions. You are expected to implement the bold-faced function for each cache.

access: This function is a read or a write request from the processor to the cache. The processor will present a virtual address, and whether the request is a read or a write. You will then update the appropriate cache metadata and statistics according to the cache type, following a least recently used (LRU) replacement policy.

dumpResults: This function will print out the cache statistics that access has been tracking. We will call this function at the end of the Pin run from the code that we have provided you. **Don't modify this code** or you'll break our testbenches, which will be tragic for the correctness portion of your lab grade.

The lab assumes 32-bit Virtual Addresses, and logPhysicalMemSize-bit Physical Addresses. The following global function will be used to translate between virtual and physical addresses:

> getPhysicalPageNumber: This function translates virtual page numbers from Pin into 'physical' page numbers, akin to the translation look-aside buffer and page table. It produces physical page numbers of size (logPhysicalMemSize - logPageSize). You should note that different virtual addresses may map to the same physical address (aliasing). *You do not need to worry about solving this in your implementation.*

In this lab you will implement the above boldface function which will track the cache metadata, thereby simulating the behavior of a real cache. The metadata that you will track includes the cache valid bits, the cache tags, and information for the least recently used (LRU) replacement policy. Recall that least recently used replacement means that when a cache block needs to be evicted, the block that was read or written farthest in the past will be chosen for eviction.

Notice that the `CacheModel` base class has a number of constructor parameters, such as associativityParam, logNumRowsParam, and logBlockSizeParam. You will use these parameters to implement your functions in such a way that multiple cache sizes and associativities can be tested rapidly. We use the logVariable naming convention to denote that the parameter is given as a base 2 logarithm of the actual size, thus if logBlockSizeParam is 4 then the block size is 2^4 = 16 bytes. Notice that our instantiations of the cache classes that you will implement use Pin command line parameters to obtain parameters. Thus, you can test your caches with many configurations, even though our public testbench will only examine a trivial configuration. The baseline caches that you will be implementing will use the appropriate number of low order bits of either the virtual or physical index as the hash function for accessing the rows of the ways. You should use the scheme in the above diagram to organize your cache.

Lab Goal

You will implement **three cache models**, one that is virtually indexed and virtually tagged (VIVT), one that is virtually indexed and physically tagged (VIPT), and one that is physically indexed and physically tagged (PIPT). Each cache will be an extension of the base CacheModel class. Each cache model will track a variety of statistics. You will track the total number of read accesses, the total number of write accesses, the number of read hits, and the number of write hits. Your labs will be evaluated for accuracy based on these statistics on both the public testbench (which you can run with `lab1test.pl`) and private testbench.

Note that the only parts of the code you have to edit for this lab are the analysis routine of the Pintool -- the instrumentation routine (which inserts the analysis routine cacheLoad() and cacheStore() before every memory load or store) has already been defined for you. In addition, the Fini function will be called at the end of program execution and will produce an output file that stores the number of requests and cache hits for reads/writes. You are welcome to make temporary changes to the Fini function to debug your Pintool. **However, do not modify the Fini function for the final submission, or you**

will not pass our test benches. There should also be no additional print statements in the Pintool that generate unnecessary outputs.

x86 is a rather quirky architecture. Recall that some architectures like MIPS allow only 4B word-aligned memory accesses. Aligned memory accesses means that the bottom bits of the address of the requested data must be zero. Thus, the address of a four-byte access must always end in 2'b00. X86, however, allows unaligned memory accesses, which will introduce some potential inaccuracy into our cache models. We will ignore handling unaligned memory accesses by assuming that the appropriate bottom bits of the access address are zeroed out.

Although your solution will not be graded on its performance in terms of wall clock time, you should note that your Teaching Assistant is impatient. The TA solution runs the sample testbench in less than 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 that time, we will kill it and assign a grade based on progress to that point. Do not write horrendously inefficient code: it makes kittens sad.

We have provided you with an additional script to enable you to run all the SPEC binaries for different cache configurations from command line.

```
% ./lab1test_param.pl -b <logBlockSize> -r <logNumRows>
-a <associativity>
```

When you have completed the lab to your satisfaction, submit your changes to your git repository. The deadline for submission is 23:59:59 EDT 29 September 2023. We'll grade whatever code you have committed and pushed by the deadline. **No Late Submissions will be accepted!** Seriously.

Lab Questions

Your response to the lab questions should be typed in lab1questions.pdf (or lab1questions.docx) in the lab1handout directory. Some questions may require coding, and as such should not be put off until the last minute. All figures and data necessary to understand your report must be included in it.

1. Hit and miss. Using your physically indexed, physically tagged cache model, starting from a base configuration (Rows = 512, Block Size = 4 bytes, Associativity = 1), vary the number of rows, block size, associativity, and capacity of your cache models. Explain any general trends in hit and miss rate that you observe (show graphs). Are there significant differences between the three cache models? Why?

2. SPEC interrogation.

a. Using your physically indexed, physically tagged (PIPT) cache model, determine the size of the working set (lookup what it means) for the SPEC2000 benchmarks, justifying your answer with a graph. Are the working sets of the SPEC benchmarks intended to fit in a processor cache (lookup the typical sizes of L1 these days)? Why or why not?

b. Again, using the PIPT cache model, do you notice a difference between the SPECINT and SPECFP programs in terms of hit and miss rates? Explain why this difference occurs.
c. Would you choose different cache configurations (block size, associativity and number of rows) given the same cache size, for floating point programs and integer programs? Which configuration(s) would you choose and why? (Remember that an associative cache takes more chip area and power than a direct mapped cache of the same capacity).

3. Make the common case fast. We asked you to implement three cache models. A potential fourth cache model is a physically indexed, virtually tagged cache. Does this cache configuration make sense? Explain why or why not. Take timing issues in to account.

4. Make the uncommon case correct. For this problem, ignore the case when multiple virtual addresses map to a single physical address. Although our baseline test bench only behaves as if the simulated machine has a single process running, real machines almost always have more than one process at a time. Uh-oh, it seems like one of the cache models may not work in the presence of multiple processes. Which one, and why? Explain what modifications are necessary to make this cache model work in the presence of multiple processes.

5. You assume too much. We assumed that all memory accesses were aligned like in the case of MIPS. Design a simple experiment to determine how many unaligned memory accesses occur in the SPEC benchmarks. What do you observe? Does our assumption make sense?

When you have answered these questions to your satisfaction, put them in a file called lab1questions.pdf (or lab1questions.docx) in your lab1handout directory, then run the following to add, commit, and push them.

```
% git add lab1questions.pdf
% git commit -a -m "Lab 1 Questions Check-In"
% git push origin master
```

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

Lab Grading

10%: Submission compiles
20%: Submission passes public test bench
20%: Submission passes private test bench
50%: 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 on Piazza, using a private Piazza post if sharing any information about your code. This will help to ensure prompt fixing of any issues that may arise.

Additional Resources:

https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html - Pin home page
https://software.intel.com/sites/landingpage/pintool/docs/98612/Pin/doc/html/index.html - Pin User Guide
https://help.github.com/articles/git-and-github-learning-resources - Git learning resources
https://git-scm.com/book/en/v2 – ProGit ebook