Cache simulation

The goal of this lab is to explore the operation of various caches using the Intel Pin tool.

Get start

To get started, download pin_cache.tgz and extract to the root of your lab folder.

```
tar xvf ~/pin_cache.tar
cd pin_cache
```

These lines create a folder `pin_cache`, this is where you will submit your files.

Create a README file in the root of your project folder. Put your name and your partners' name in the file, Also, use this file for any comments you want to make and to answer the questions given in this lab.

Pin is a binary instrumentation tool that can call out to user-supplied object files. We will be creating our own cache module that Pin will call. As to Pin's working environment, **please check the 'Get-Start' part of Branch Predictor's Guide**.

For now, let's run a simple test to make sure Pin is working. Run the following commands from your `pin_cache` project folder. Note that `PIN_ROOT` shall be your root directory of pin and `obj-intel64` may have to be changed to `obj-ia32` which depends on the version of pin you use.

```
cd ~/pin_cache
export PIN_ROOT=~/"the root directory of pin"
$PIN_ROOT/pin -t $PIN_ROOT/source/tools/ManualExamples/obj-intel64/inscount0.so -o inscount0.log -- tar zcf data1.tgz data1
```

This should run for a bit and then exit without outputting anything. What this is doing is executing Pin and asking it to use to `inscount0.so` tool provided with Pin and store the output to inscount0.log. The tool `inscount0` simply counts the number of machine instructions that were executed by the target (in this case, the tar zcf command). When Pin executes, it instruments the command given after the `--`, in this case, `tar zcf data1.tgz data1`. The file data1 is a 16 MB file of random noise. `cat inscount0.log` and you should get something like the line below.

```
Count 1491034
```

Interestingly, the number you get won't be exactly the same and if you run this a few times you will see some variation. This suggests that the execution of tar/gzip is not entirely deterministic, but we'll leave that for later exploration. If you've gotten this far, Pin is working! Take a minute to count instructions on a few programs of your own. Many more interesting examples are provided in [the Pin documentation].

On to some cache work

Our goal is to explore cache access in real programs using Pin. Since Pin is just an instrumentation tool, it doesn't have a built in cache simulator. No worry, I have one you can use. In the subfolder `cache` there is a cache simulator written in C. Compile this and run a test as shown below. The Pin `makefile` requires the path to Pin be supplied on the command line.

```
cd cache
make PIN_ROOT=$PIN_ROOT
```

This will create `pincache.so` in `./obj-intel64` (or `./obj-ia32`). This is the code that sets up Pin to pass memory accesses to our simulated cache. The source is in `pincache.cpp`. You do not need to modify this file just yet, but feel free to browse around and see how it works. Basically, for every instruction that accesses memory, we insert a call to `simulate_access` with each memory location that the instruction accesses. Also note the command line arguments and their default values at the top of `pincache.cpp`.

Test the simulated DM cache

We can use Pin to run our tool on tar/gzip and display the output as below.

```
$PIN_ROOT/pin -t obj-intel64/pincache.so -- tar czf
data1.tgz ../data1
cat cache.out
```

You should get something similar to the output below.

```
Cache statistics:
  Writes: 160394
  Write Misses: 1281
  Reads: 369779
  Read Misses: 3716
  Misses: 4997
  Accesses: 530173write
  Miss Ratio: 0.942523%
```

You can modify the size of the cache with the `-s x` parameter (after `pincache.so`), where `x` must be a power of 2.

Question 1

Copy and complete this table in your README file:

```
1) DM miss ratio on `tar czf ...` with the given total cache size with 64 byte blocks
   4K
  32K
 256K
2048K
```

Implement an associative cache

Modify `cache.c/h` to support the `-a` command line option to specify cache associativity. Assume LRU replacement. You should support up to and including a fully associative cache (e.g., for a 32KB cache with 64B lines, `-a 512` is fully associative).

The programs `test0` and `test1` are provided to test your cache (without involving Pin). They can be built using the provided `makefile` (e.g., `make PIN_ROOT=$PIN_ROOT test0`).

The file `test0` tests the direct mapped cache and will pass with the provided code.

The file `test1` tests a 2-way associative cache and will fail with the provided code.

After supporting 2-way associativity, `test1` should now pass.

After you get `test1` working, create `test2` to test a fully-associative 512B cache with 64B blocks. Be sure to check the replacement policy as well. Be sure to add `test2` to your hand-in.

Hints: You might want to enable debugging in `cache.c`. On line 233, change this to `self->dbg = 1;` and re-compile to see each and every access to your cache. There are also many hints in the comments of `cache.c` to help guide you.

Question 2-5

Copy and complete these tables in your README file:

```
4) 2-way associative miss ratio on `tar czf ...` with the given
total cache size with 64 byte blocks and LRU replacement
   4K
  32K
 256K
2048K

5) 4-way associative miss ratio on `tar czf ...` with the given
total cache size with 64 byte blocks and LRU replacement
   4K
  32K
 256K
2048K

6) 8-way associative miss ratio on `tar czf ...` with the given
total cache size with 64 byte blocks and LRU replacement
   4K
  32K
 256K
2048K

7) fully associative miss ratio on `tar czf ...` with the given
total cache size with 64 byte blocks and LRU replacement
   4K
  32K
 256K
2048K
```

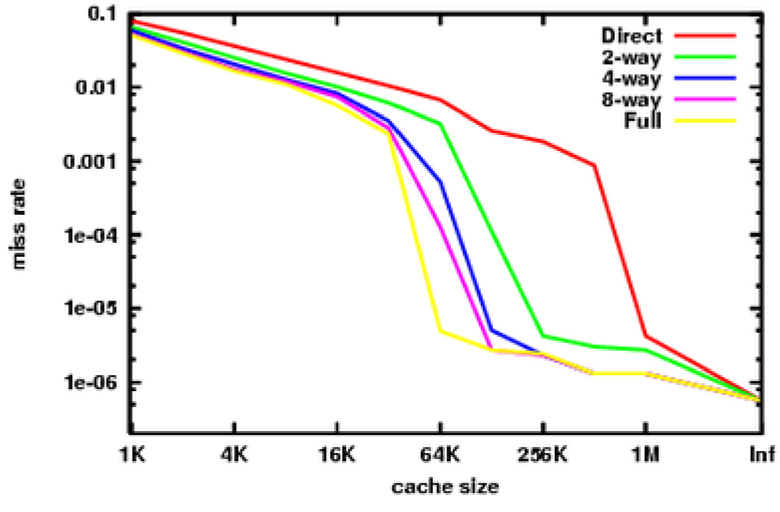
Implement other replacement policies

Now let's explore replacement policy. Add a new `KNOB` to `pincache.cpp`, to accept an argument called `replace` on the command line that will configures the replacement policy. You need to support `-replace LRU` (the default), `-replace RANDOM`, and `-replace FIFO`.

Create the file `test3` to verify these replacement policies.

Plotting

Create a script (or you can test all the case one by one) to call your Pintool and iterate through all combinations of cache size (4K, 32K, 256K, 2048K), associativity (DM, 2-way, 4-way, 8-way, fully), and replacement strategies (LRU, RANDOM, FIFO) and create a line graph of the results. Plot cache size on the X-axis and miss rate as the Y-axis. Create one graph for each replacement strategy and create a series (line) for each associativity. Save all plots as `.png` images. You should create `results-LRU.png`, `results-RANDOM.png`, and `results-FIFO.png`. Where each plot has 5 data series (DM, 2-way, 4-way, 8-way, and fully associative). One possible way to create your plot is with gnuplot (or Excel, or any way you prefer). Each of your 3 graphs should look something like the one below:



Submit

You have likely modified the functions in `cache.c`. Go back and make sure **all** test cases still work. You might have to supply some extra parameters to the cache (i.e. replacement policy). When the test cases all work, pack all the files and folders and email them to ***liang.cao.24.31@gmail.com***

Bonus

2-level Cache

Copy your files from `~/pin_cache/cache` to `~/pin_cache/mcache`. Modify the files in `mcache` to implement a 2-level cache. This cache should just create two instances of `avdark_cache_t`. One for L1, the other for L2. These caches may have different parameters. You will need to modify `avdc_access` to return the value of `hit`, to decide if L2 should be accessed. You may decide to make your caches inclusive, or exclusive (or add a `KNOB`!). Create `test4` to verify your 2-level cache.

Grading

100 points total plus the possibility of 20 bonus points.

1. [5] q1: 5 points
2. [15] test1 works: 10 points
3. [15] test2 created and checks fully associative cache
4. [20] q2-q5: 5 points each
5. [15] test3 checks LRU, RANDOM, and FIFO replacement, 5 points each
6. [30] result.png looks good
7. [20 bonus] mcache created, test4 checks [BONUS]