As described in class, timing side channels are a feature of shared caches in which a process may inadvertently *leak* information about which data it uses based on the state of data in a cache. One of the key mechanisms that an adversary may take advantage of to control for what information is leaked is to control the contents of the cache state. In particular, an adversary may fill the cache with dummy data during a *prime and probe attack* to ensure that a victim's sensitive data is evicted by *priming* the cache state prior to later *probing* the cache to see if the sensitive information is there.

One of the open questions from class was how an adversary goes about ensuring that a victim's data is evicted from the cache. Ultimately, this is achieved by reverse engineering the caching strategies and using this learned information to fill the cache. In this lab, you will be writing programs to help perform this reverse engineering. If you then wanted to use this information to perform an attack, you could use the learned context about the cache construction to selectively evict sensitive data from the cache state, and then later query that cache to see if the data exists. This would provide context as to the victim's behavior with respect to using that sensitive shared data.

1 Getting Started

Start by getting the materials for this lab from the course page. From inside your cache assignment stencil, you can do so by running:

```
wget https://cs.pomona.edu/classes/cs181ca/labs/lab07-tests.zip
unzip lab07-tests.zip
mv lab07-tests/configs/* configs/example/gem5_library/
```

You should not need to recompile gem5 for this lab.

2 Learning Cache Sizes

Suppose we are running on an unknown platform (e.g., our attack is running on hardware owned by the victim). If this is the case, then it is unlikely that we would know any of the configuration details about the victim's memory system (e.g., cache size, associativity, etc). As an adversary, our goal is to control the contents of the data in the cache. Therefore, to know how much dummy data we may want to load into the cache, we should first learn how big the cache is!

If we want to learn about the configuration of the memory system, we should start by writing an application that goes to memory. Because arrays are essentially blobs of data in memory, we can do so by creating a big array, and accessing it in sequence. Look in the `lab07-tests/src` directory. Notice that, in this directory, there's a sample file (`test-16kB.c`) to get started, which we can build by calling `make test-16kB` from inside the `lab07-tests` directory. The resulting binary goes to the `lab07-tests/bin` directory.

For completeness, the `test-16kB.c` file looks like the following:

```c
int
main(int argc, char **argv)
{
    unsigned long long BLOCK_SIZE = 64;
    unsigned long long SIZE = 16 * 1024;
    char arr[SIZE];

    for (unsigned long long i = 0; i < SIZE; i += BLOCK_SIZE) {
        arr[i]++;
    }

    return 0;
}
```

2.1 Measuring Performance

If we want to see how long our program will take to execute, we can run it on our configured platform and measure the performance! Fortunately, gem5 does all of the measuring for us, so we don't need to add timers or anything to our source. To run, we can use the command `./build/X86/gem5.debug configs/example/gem5_library/lab07-secret-config1.py --benchmark=lab07-tests/bin/test-16kB`. This allows us to specify the binary to run by passing it via the `--benchmark` flag whereas the hardware configuration is specified with the Python configuration script. Note, there are three candidate secret configuration scripts that you can use!

Once we start running our workload, we can analyze the outputs to determine the "performance" (or runtime speed) of our run. We can do this by opening the `m5out/stats.txt` file, but our lives will be much easier if we automate this process. Let's write a Python script that prints out the execution time for our run!

```python
with open(f'm5out/stats.txt', 'r') as f:
    lines = f.readlines()
    time = 0.
    for line in lines:
        if 'simSeconds' in line:
            time = float(line.split()[1])

    print(f'execution time:\t{time}')
```

If this script is called `time-per-access.py`, then we can get the execution time printed out cleanly by running `python3 time-per-access.py`.

2.2 Inferring Cache Size from Performance

Unfortunately, a single performance metric in isolation does not provide enough context to be able to figure out the cache size. If we wanted to reverse engineer the size, we could do so by using the motivating goals for using caches to infer what is going on underneath the hood! In particular, we know that caches are designed to take a *small amount of data* and ensure fast access to it. Therefore, if we access enough data, all of a sudden our performance should be really slow – your goal is to write a set of programs that measure and study at what point our performance starts to get slow! (solution on next page)

Hopefully, you created a bunch of workloads for which the array size gets increasingly large. After you do this, you will want to run each of these input sizes through gem5 and save the outputs in different directories (e.g., `./build/X86/gem5.debug -d config1-test16kB configs/...`). From here, we can modify our `time-per-access.py` to print the runtime for each of the outputs.

What do you notice about the outputs?[1] Is this a function of the cache behavior or something else? As it turns out, our application looping over bigger and bigger arrays will run slower because there is more work to do! If we want to know the performance on a per-instruction basis, we should use a different metric (cycles per instruction). Let's modify our `time-per-access.py` script.

```
for outdir in ['all', 'of', 'your', 'outdirs']
with open(f'{outdir}/stats.txt', 'r') as f:
    lines = f.readlines()
    cycles = 0.
    insts = 0.
    for line in lines:
        if 'simTicks' in line:
            cycles = float(line.split()[1])
        if 'simInsts' in line:
            insts = float(line.split()[1])

    print(f'{outdir}:\t{cycles / insts}')
```

When we make these modifications, what happens this time around?[2] Why do you think this is?[3] Try extending your workload to see if you can visibly reduce the CPI (solution on next page).

[1] They all increase monotonically.

[2] The CPI is more or less the same for all of our workloads.

[3] Applications only get to have faster memory accesses on reuse! Our workloads currently only access every element in the array once.

Our new version of `test-16kB.c` could look like:

```c
int
main(int argc, char **argv)
{
    unsigned long long BLOCK_SIZE = 64;
    unsigned long long SIZE = 16 * 1024;
    char arr[SIZE];

    for (int num_interations = 0; num_iterations < 100; num_iterations++) {
        for (unsigned long long i = 0; i < SIZE; i += BLOCK_SIZE) {
            arr[i]++;
        }
    }

    return 0;
}
```

How does your CPI change if you add more ... TO BE CONTINUED