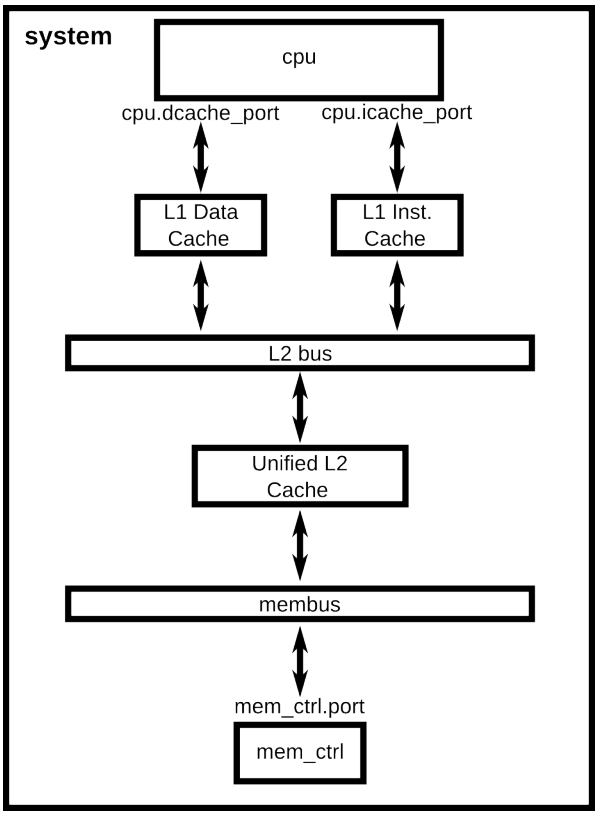
CSE 5600 Computer Systems Architecture I

Assignment 3, due Nov. 6, 2026

In this lab assignment we will explore using the `gem5` simulator to look at cache simulation and program statistics.

1. In this exercise, we will be simulate the following system, which was built during assignment 2:



   Just like before, copy your configuration files from assignment 2 into a new folder `hw3`.

   What we want to test in this assignment is how different cache subsystems affects the execution of programs. In this case we want to see how different cache associativity and cache sizes impacts the running time of our programs and calculate some statistics about what types of cache activity we are seeing. Intuitively, this means that we will be adding a configurable cache to our `cache.py` file. Open it up and import

```
import gem5.components.cachehierarchies.classic.
private_l1_private_l2_cache_hierarchy as private_l1_l2
```

Then we need to change the function that we currently have. Right now it statically constructs a cache hierarchy, but to do these experiments we want to be able to set the size of the l1d cache. First, update the function to take in a single parameter `args`. Then inside the function add these lines

```
class CustomL1DCache(private_l1_l2.L1DCache):
    def __init__(self, size=args.l1d_size):
        super().__init__(size=size)
        self.assoc = args.l1d_assoc

private_l1_l2.L1DCache = CustomL1DCache
```

And then finally just swap what is returned from the function with:

```
return private_l1_l2.PrivateL1PrivateL2CacheHierarchy(
    l1d_size=args.l1d_size,
    l1i_size="16KiB",
    l2_size="256KiB",
)
```

And that's it. We can add a few new parameters to the `args.py` file as well to make running the experiments easier:

```
parser.add_argument(
    "--l1d-size", "--l1d_size",
    dest="l1d_size",
    default="16KiB",
    type=str,
)

parser.add_argument(
    "--l1d-assoc",
    "--l1d_assoc",
    dest="l1d_assoc",
    default=8,
    type=int,
)
```

Once that's done, you're ready to do some simulations. Copy your original entry-point script, called `03_minor_single_double.py` in the last assignment, to `assignment3.py`

and make sure to pass in `args` to the `make_cache_hierarchy` function call. Once you've done that verify your simulation is working correctly by running this command:

```
$GEM5/build/RISCV/gem5.opt --outdir="queens" assignment3.py --prog="queens"
--l1d_size="1kB" --l1d_assoc="2"
```

Include a screenshot of the console output in your writeup.

2. Now that we verified we have a working system let's change some parameters and get some measurements! Change your simulation so that the following parameters are implemented and run a simulation for each.

   Create a table for using the `RISCV` ISA of the following parameters and record the value of the parameter for each cache configuration. Pick a parameter (an interesting one that is either listed here or that you've found on your own) and create a graph making it a function of the cache size. As an answer to this question show the graph and table that you have created.

L1D$ Association	L1D$ Size
2	1 kB
2	2 kB
2	4 kB
2	8 kB
2	16 kB
4	1 kB
4	2 kB
4	4 kB
4	8 kB
4	16 kB

Parameters to record.
Instructions Committed
Average Gap Between Requests
L1D$ Overall Hits
L1D$ Number of replacements
L1D$ Overall Miss Rate
L1I$ Overall Miss Rate
L2 $ Overall Miss Rate
Number of CPU Cycles

3. Pick two configurations from the previous problem and calculate their CPI as a function of the number of instructions and total number of CPU cycles. Show your work! Compare their CPI's. Is there a difference between the CPI of these two configurations? Why or why not?

4. Additionally, we are going to try to classify all the types of cache misses this program encounters during a particular execution. To classify the cache misses encountered while running this program into cold, capacity, or conflict misses, you will need to execute the program multiple times using different cache configurations. For instance, to identify cold misses exclusively, you should increase both the capacity and associativity to very high values. This adjustment ensures the absence of capacity and conflict misses. By conducting various tests, you can create a table correlating each type of miss with a number from the overall misses.

   For the `RISCV` ISA, with a 1 kB L1D$ size with 2 associativity, perform appropriate experiments to classify the the L1D$ misses as cold, capacity, or conflict misses.

   Note: For some of the possible configurations, GEM 5 may crash during simulation or simply disallow running with those parameters. To avoid at least part of these, only

increase the associativity to a max of 8 and size up to 8kB. This is enough space to see what you need to to classify the misses.

For submission, produce a table that includes an entry for each miss type, along with the count of misses from the overall miss tally that correspond to each specific type. Additionally, you should document the test conditions used to determine these numbers, specifying the L1D$ size and the L1D$ associativity. An example table is provided below (the chosen size and associativity are just for illustration).

Miss Type	Total Misses	Configuration Used To Discover This
Total	1234	L1D$ Size: 1kb — L1D$ Association: 2
Capacity	...	...
Conflict	...	...
Cold	...	...