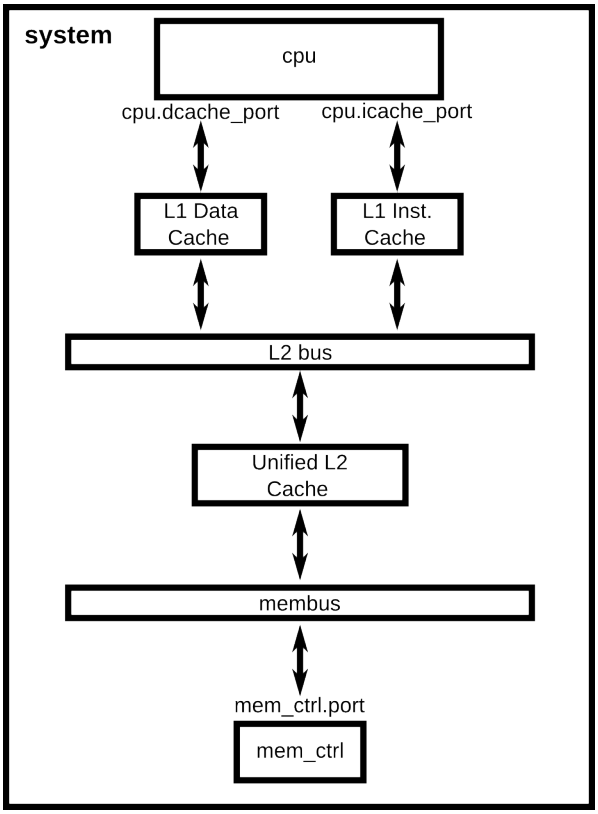
CSE 560M Computer Systems Architecture I

Assignment 2, due Friday, Oct. 11, 2024

In this lab assignment we will explore using the `gem5` simulator to look at branch prediction. To make the performance reasonable, we will include a cache subsystem. The cache subsystem will be explored in more detail in the next assignment.

1. In this exercise, we will be creating the following system:



This may look somewhat similar to your previous system but with some extra building blocks added. Instead of a simple unified memory system there is now a hierarchy of caches for the CPU to query for memory. Here, we've added a L1 data and instruction cache along with a unified L2 cache to the configuration file. These caches will be extensions of the `Cache` class and will be instantiated in a similar way to the previous lab.

Of course, this base class does not have every parameter instantiated for us so lets walk through instantiating one of these SimObjects. We'll start with the L1 instruction

cache to begin and leave the rest as an exercise. First we'll need to create a `Caches.py` file alongside our `gem5` configuration file from last lab. To start we'll need to create the file `Caches.py` by running the following command in a `cse560m/hw2` folder:

```
touch Caches.py
```

Then opening the file in the text editor of your choice begin by typing out the code to import the gem5 building blocks.

```
from m5.defines import buildEnv
from m5.objects import *
```

Then, on the next line, we'll start by declaring the class `L1Cache` and setting some default parameters:

```
class L1Cache(Cache):
     assoc = 2
     tag_latency = 2
     data_latency = 2
     response_latency = 2
     mshrs = 4
     tgts_per_mshr = 20
```

Now we need to define an initialize function and connections for the cache:

```
     def __init__(self, options=None):
         super(L1Cache, self).__init__()
         pass
     def connectBus(self, bus):
         self.mem_side = bus.slave
     def connectCPU(self, cpu):
         raise NotImplementedError
```

NOTE: These functions are within the scope of the `L1Cache`, take note of the indents.

Now we can move on to actually defining a cache that will be used. On the next line we will define the actual L1 instruction cache.

```
class L1ICache(L1Cache):
     is_read_only = True
     writeback_clean = True
     size = '16kB'

     def __init__(self, opts=None):
         super(L1ICache, self).__init__(opts)
         if opts.l1i_size:
             self.size = opts.l1i_size
         if opts.l1i_assoc:
             self.assoc = opts.l1i_assoc

     def connectCPU(self, cpu):
         self.cpu_side = cpu.icache_port
```

A couple things to note with this configuration: This cache is defined as read only meaning that we can't write values back to the cache, which makes sense as we wouldn't want to corrupt our instructions. Next, the `writeback_clean` option is set to true denoting that writebacks will happen when evicting clean lines (As a fun exercise think about why this is specified here). After defining the size we add options for size and whether or not the cache is associative. Following that, the initialize function is defined which will load in the size and association options for the cache. Finally, the connections are defined stating that the connection between itself and the CPU is through the CPU's icache port.

Now, starting with what we've defined here, implement a L1D cache and create a L2 cache which will be its own extension of the `Cache` (not `L1Cache`) class.

The member functions for the L2 cache should include the following:

```
     def connectCPUSideBus(self, bus):
         self.cpu_side = bus.master
     def connectMemSideBus(self, bus):
         self.mem_side = bus.slave
```

The specifications for both L1 D- and L2 Cache are as follows:

	L1 Data Cache	L2 Cache
is_read_only	False	False
writeback_clean	False	False
size	64kB	256kB
assoc	default	8
tag_latency	default	20
data_latency	default	20
response_latency	default	80
mshrs	default	20
tgts_per_mshr	default	12
Connections	cpu.dcache_port	CPUSideBus:master MemSideBus:slave

And for both caches add the option to specify their size and associativity as we did in the L1 Cache.

2. Next, we need to alter the CPU model so that it can support the branch predictors. To start, copy your configuration file from the first assignment, `x86_vs_arm.py`, into a new file with some meaningful name, in this case `assignment2.py`.

   ```
   cp ../hw1/x86_vs_arm.py assignment2.py
   ```

   From there open your newly copied file in a text editor of your choice.

   First we will add the ability to set the type of branch predictor, its size, and the number of bits in the counters on the command line. Add the following after the existing `parser.add_option` command.

   ```
   parser.add_option("--bp", type="str", default=None)
   parser.add_option("--bp_size", type="int", default=None)
   parser.add_option("--bp_bits", type="int", default=None)
   ```

   and add
   `bp = options.bp`

   after parsing the arguments with `parser.parse_args()` and setting `program` just a few lines down.

   Next you'll need to **remove** the ISA specific clock settings.

   ```
   if isa == "x86":
       system.clk_domain.clock = '1GHz'
   elif isa == "arm":
       system.clk_domain.clock = '1.2GHz'
   ```

   And replace with:

   ```
   if options.clock_freq:
       system.clk_domain.clock = options.clock_freq
   else:
       system.clk_domain.clock = '1.2GHz'
   ```

   For the next two assignments, we will only use the Arm ISA.

The `system.cpu = TimingSimpleCPU()` line gets replaced with the following

```
if bp == "LocalBP":
    system.cpu = MinorCPU(branchPred=LocalBP())
    system.cpu.branchPred.BTBEntries = options.bp_size
    system.cpu.branchPred.localPredictorSize = options.bp_size
    system.cpu.branchPred.localCtrBits = options.bp_bits
elif bp == "TournamentBP":
    system.cpu = MinorCPU(branchPred=TournamentBP())
    system.cpu.branchPred.BTBEntries = options.bp_size
    system.cpu.branchPred.localPredictorSize = options.bp_size
    system.cpu.branchPred.localCtrBits = options.bp_bits
    system.cpu.branchPred.globalPredictorSize = options.bp_size
    system.cpu.branchPred.globalCtrBits = options.bp_bits
else:
    system.cpu = MinorCPU()
```

Here, the branch predictor can be set to `LocalBP`, which acts as the simple branch predictors initially described in class (if `bp_bits`=1, it remembers the last value; if `bp_bits`=2, it is a 2-bit counter). Or it can be set to `TournamentBP`, which is the hybrid predictor described in class (the combination of the simple predictor and the correlated predictor). The `bp_size` parameter sizes all of the relevant tables (including the BTB).

3. Now lets add the caches that we've created to our system configuration file.

   First we will need to **remove** the following commands from the last assignment since we are going to connect the L1 caches to the L2 cache.

   ```
   system.cpu.icache_port = system.membus.slave
   system.cpu.dcache_port = system.membus.slave
   ```

   Now we can import your newly defined caches in the script.

   ```
   from Caches import *
   ```

   You'll need to use the `parser.add_option` command to add all of the options you added in your caches; otherwise, `Python` will complain. You should also add an option for varying the CPU clock speed. Note that the clock frequency and cache size will be of type `str`, and the cache associativity will be of type `int`.

   Now we will start to connect the modules together, first instantiate the L1 caches using the following lines of code:

   ```
   system.cpu.icache = L1ICache(options)
   system.cpu.dcache = L1DCache(options)
   ```

   Then connect them to your cpu using the following lines:

   ```
   system.cpu.icache.connectCPU(system.cpu)
   system.cpu.dcache.connectCPU(system.cpu)
   ```

Now we need to create a bus to connect our L1 caches to L2, however, it only has one port so we need to create a bus to connect the three along with control signals from the CPU.

```
# Create a memory bus, a coherent crossbar, in this case
system.l2bus = L2XBar()
# Hook the CPU ports up to the l2bus
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)
```

Then create the L2Cache in the system and connect it's `CPUSideBus` to the `l2Bus` and it's `MemSideBus` to the `membus` created in the last assignment.

```
system.l2cache = L2Cache(options)
system.l2cache.connectCPUSideBus(system.l2bus)
system.l2cache.connectMemSideBus(system.membus)
```

You should now have a complete system with a working branch predictor and L1 & L2 cache! In this lab we are going to use the program `queens` to test the system.

Now in your `hw2` directory, test your `assignment2.py` configuration file using the command:

```
$GEM5/build/ARM/gem5.opt --outdir="queens_tournament" assignment2.py \
    --prog="queens" --bp="TournamentBP" --bp_size="8192" --bp_bits="2"
```

Note that the \ in the above command is simply indicating that it runs onto the second line.

Include a screenshot of the console output in your writeup.

4. Now that we have a working system let's change some parameters and get some measurements. While we have the ability to vary the parameters of the cache, we'll save that for next time. In this assignment, our interest is in the performance impact of the branch predictors.

   We can independently set the following things:

Parameter	Label	Possible Values
branch predictor	`bp`	`LocalBP`, `TournamentBP`
table size	`bp_size`	16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192
counter bits	`bp_bits`	1, 2

   However, not all parameter value combinations make sense. With the tournament predictor, the counter bits should always be set to 2.

   Unfortunately, the low-level details of the branch predictors are sufficiently different (both from each other and from the simplified discussion given in class) that the various internal statistics associated with branch prediction cannot fairly be compared with one another. So, we will rely on how they impact CPI. Fortunately, the new processor model (`system.cpu`) shows us CPI directly in `stats.txt`.

   For this assignment you are to execute 30 separate simulation runs:

- All 10 table sizes for the local predictor with a 1-bit counter. A label for this set is `Loc1`.
- All 10 table sizes for the local predictor with 2-bit counters. A label for this set is `Loc2`.
- All 10 table sizes for the tournament predictor with 2-bit counters. A label for this set is `Tour`.

These simulations are all short, so they shouldn't take very long to run. You do not need to include screenshots for each run, the example run above is sufficient. Do, however, check the generated `config.ini` files to ensure that you really are running the simulations you think you are. The parameter labels in the configuration file are those set in the `.py` file above.

5. For each set (`Loc1`, `Loc2`, and `Tour`), plot CPI as a function of table size. To improve the readability of the graphs, you might make the y-axis something larger than zero. You can put all three plots on one graph (if they are sufficiently separate to distinguish each point), or you can use three separate graphs, your choice. Describe (in a paragraph or so) what you can conclude from this data. Are the results what you expected? Any surprises?