1 Getting Started with gem5

1.1 Downloading and installing a clean version of gem5

For this I recommend using the gem5 documentation (`https://www.gem5.org/documentation/learning_gem5/part1/building/`). For the most part, the documentation on their site is pretty good, and some parts are kept relatively up to date. I recommend combing through their "Learning gem5" module just to get a sense of what the simulator is and how it works so that you have a foundation of what needs to happen when you want to make a change for research.

Let me know if you have any questions, but make sure that you can compile gem5 (`scons build/<ISA>/gem5.<optimization level>`) first before moving onto the next steps. This step can be slow, but you can make it run faster by using parallel compilation (adding "`-j<num procs>`"). I recommend that you do this either in the course VM (which you can access calling `ssh <username>@itbdcv-lnx04p`) or by running gem5 in a Docker container. There is more information about using these platforms in the **Docker Guide**.

1.2 Running a Program in gem5

In gem5, there are two modes of execution. SE mode is the faster one, and it relies on your host machine to emulate system calls (hence the name). As such for this to work, you need to have gem5 compiled for the same ISA as your host machine (i.e., Intel CPUs need to have gem5 compiled with "`scons build/X86/gem5.<debug,opt,fast>`").

Once that is done, we can start running a program in gem5. To do so, we will create a bash script in the base gem5 directory called "`run.sh`" (or whatever you want).

```
#!/bin/sh

CUR_DIR=$(pwd)

$CURR_DIR/build/<your ISA>/gem5.opt -d $CURR_DIR/results/hello \
    $CURR_DIR/configs/learning_gem5/part1/simple.py
```

We can run this program by calling `./run.sh` and it will simulate a real hardware platform running the `hello` binary! The simulation output and results will be visible in the `$CUR_DIR/results/hello` directory. Take a peak at the `stats.txt` file and look at all of the metrics we can collect!

We are relying on the `simple.py` configuration file. Let's look at this file. In this configuration file, the binary to simulate is specified to be the `hello` executable in `tests/test-progs/hello/bin/x86/linux/hello`. If we wanted to run a different binary, then we can change the path to the executable binary. This means that when we run the simulator, it will use this program as the workload. Suppose we wanted to change the binary to run; we could do so by changing the path to the executable.

Let's do so in a new file called `input-binary.py` that lives in a new subdirectory within the central `configs` directory. We can create it by calling `mkdir -p configs/cs181ca/example/` and create a starting point to build our configuration from the existing script by calling `cp configs/learning_gem5/part1/simple.py configs/cs181ca/example/input-binary.py`. The first thing to add to our new `input-binary.py` script is to import the `argparse` library to handle command line inputs.

```
import argparse

...
```

From here, we can add a field that processes the command line in a reasonable way:

```
parser.add_argument(
    "--binary",
    type=str,
    required=True,
    help="Input the full path of the binary program to execute.",
)

args = parser.parse_args()
```

This allows us to set the `command` string to be the argument passed as input to the run line.

```
command = args.binary
```

And we can execute our simulator by changing our `run.sh` to now be:

```
#!/bin/sh

CUR_DIR=$(pwd)
# note: $1 will be the first argument to run the script
BINARY_PATH=$1
# a little hack to extract the last token
BINARY_NAME=$(echo $1  | rev | cut -d'/' -f 1 | rev)

$CUR_DIR/build/<your ISA>/gem5.opt -d $CUR_DIR/results/$BINARY_NAME \
    --binary=$BINARY_PATH
```

This will allow us to run `./run.sh /path/to/binary` and our simulator will execute that binary. Try it out for yourself!

As an aside, our configuration script is currently built using an old version of gem5 as the basis. In it, all components are declared individually (the processor, the memory, etc). This is no longer the standard way of building configuration scripts – recently the gem5 developers have released a "standard library" of configurations with a cleaner interface and syntax. You can perform the same exercise using `configs/example/gem5_library/armhello.py` as the starting configuration to copy. In this file, the simulated components are primarily added via the "`Board`" interface, which takes a processor, memory and cache hierarchy as input. Then, the binary is set to the board with a call to `board.set_se_binary_workload()`. Unlike in the learning gem5 example, this script retrieves a file from gem5's remote resources repository and downloads the executable to simulate. You can change this behavior by importing the `BinaryResource` wrapper class as follows:

```
from gem5.resources.resource import BinaryResource

...
```

And then setting the binary to the board by calling

```
board.set_se_binary_workload(
    BinaryResource(local_path=args.binary)
)
```

Let's unpack some of what has been going on in our setup:

1. We have a shell script that runs gem5, where we define environment and command line arguments;
2. We call the gem5 binary, which we compiled with `scons` for the ISA that we wanted;
3. We called "`-d`" which is the directory to which the results will be flushed;

4. We have a method for adding customizable flags that our configuration script can parse when running the simulator;
5. We called a Python script called "input-binary.py" that acts as a simulator configuration which declares the components to use (`SimObjects` in the C++ source, we will go through these later) – you will not need to understand the linking between Python front-end and C++ backend to use gem5;
6. The configuration files declare a processor with a particular "CPU type" (`X86TimingSimpleCPU` or `SimpleProcessor(cpu_type=CPUTypes.TIMING)`) – there are three processor types that we will use in this class: `Atomic` (not timing-accurate, but fast for setting checkpoints), `Timing` (timing-accurate, in-order execution), and `O3CPU` (sometimes "`DerivO3CPU`", slow execution, timing accurate out-of-order execution);
7. The configuration script declares the cache hierarchy, of which there are several ways to configure its implementation;
8. We have modified the script to use whatever binary you want to specify, there are samples in the `tests/test-progs` directory!

The exmaples that we've performed so far are described as "syscall emulation mode" (SE mode), where the operating system and file system are emulated in the simulator backend. SE mode is nice for introductory examples, but the approximation of the operating system and disk with the file system is not accurate enough research simulations. In class, we will only ever need to run examples in SE mode, and starting configuration scripts will be provided to you as needed. You may want to modify/extend them to add robustness to your test suites!

2 The gem5 Back End

Now that we've gone through the steps to run gem5, let's take a deeper look at how the simulator is implemented.

Components in gem5 are implemented as `SimObject`s, an abstract C++ class that describes the fundamental execution unit for a simulated component. The abstract `SimObject` declaration can be found in `src/sim/sim_object.{hh,cc}`. The header file for this class has more information than is relevant for us to build and manage our components, so let's instead use the `HelloObject` declared and defined in `src/learning_gem5/part2/hello_object.{hh,cc}` as our working example.

The first thing to notice is that the `HelloObject` is a derived class of the base `SimObject` class. This is denoted by the line `class HelloObject : public SimObject` [1]. This means that the class will extend the functionality of the base `SimObject` class to add its own functionality (in this case, print "Hello world!" n times and then trigger a `GoodbyeObject` to execute). There are all sorts of additional fields added to this class declaration (`hello_object.hh`) in order to facilitate its function.

To execute the function of the `HelloObject`, the simulator first calls the `startup` function of every `SimObject` in the simulation by overriding the definition from the base class (which typically does nothing). This triggers the start routine in all simulations[2]. In the `HelloObject`,

[1] For more about public/private/protected inheritance in C++, see this discussion. In general, it's safe to assume that gem5 uses public inheritance.

[2] For most cases this will involve the simulated processor fetching the first operating system instruction from the known location in memory to begin the boot procedure.

the `startup` function schedules an event, which is defined as a lambda function. `event` is declared as an `EventFunctionWrapper` for the lambda function defined in the constructor (i.e., `[this] { processEvent(); }`).

Note, the primary execution method for the simulated components is to perform an "event." These events are scheduled with a timestamp (e.g., `curTick() + latency` in the case of the `HelloObject`). Some objects have self-scheduling events (like the `HelloObject`), but most events are driven by some other event happening elsewhere in the simulation. For instance, the main "event" of the pipeline stages of the pipelined processor models are triggered by the completion of the previous stage or the central processor logic.

To provide a holistic intuition for how the system works, events are placed in a priority queue (`src/sim/event_queue.{hh,cc}`) sorted by the timestamp at which they are scheduled to execute. Time advances in the simulation by popping events from the priority queue, setting the global clock (in terms of "ticks", which act like clock cycles for an arbitrary clock rate) and executing the lambda function associated with the event.

A consequence of this implementation is that debugging errors in gem5 can be tricky! The relevant state leading to an error is likely in the context of a prior lambda function rather than the current call trace. For this reason, you may need to be creative about how you debug! Be sure to revisit the **Intro to gem5** lab for tips and strategies!