MultiSim

Multiprocessing support for gem5 simulations.

The problem

The gem5 simulator is single-threaded.

This is baked into the core design and is unlikely to change due to the high cost of converting the entire codebase.

Therefore, we cannot "speed up" your work with more cores and threads.

The insight

The gem5 simulator is used for experimentation.

Experimentation involves exploring how variables of interest change the behavior of a system when all other variables are held constant. As such, experimentation using the gem5 simulator requires multiple runs of the simulator.
Multiple instances of gem5 can be run in parallel.

If not a singular gem5 process utilizing multiple threads, why not multiple gem5 processes, each single threaded?

((This is really handy for us as we don't need to worry about the complexities of multi-threading: memory consistency, etc.))

People already do this... kind of...

You could use bash scripts to accomplish this.

This is typical but not recommended.

Writing a script to run multiple gem5 processes:

1. Requires the user to write the script.
    1. Increases the barrier to entry.
    2. Increases the likelihood of errors.
    3. Requires the user to manage output files.
2. Non-standard (everyone does it differently).
    1. Hard to share with others.
    2. Hard to reproduce.
    3. No built-in support now or in the future.

A better way

MultiSim is a gem5 feature that allows users to run multiple gem5 processes from a single gem5 configuration script.

This script outlines the simulations to run.
The parent gem5 process (the process directly spawned by the user) spawns gem5 child processes, each capable of running these simulations.

Via the Python `multiprocessing` module, the parent gem5 process queues up simulations ("jobs"), for child gem5 processes ("workers") to execute.

Multisim

Multisim has several advantages over simply writing a script to run multiple gem5 processes:

1. We (the gem5 devs) handle this for you.
    1. Lower barrier to entry.
    2. Lower likelihood of errors.
    3. Multisim will handle the output files automatically.
2. Standardized.
    1. Easy to share with others (just send the script).
    2. Easy to reproduce (just run the script).
    3. Allows for future support (orchestration, etc).

Some caveats (it's new: be patient)

This features is new as of version 24.0.

It is not fully mature and still lacks tooling and library support which will allow for greater flexibility and ease of use.
However, this short tutorial should give you a good idea of how to use it going forward.

Using multisim

The main idea is that you will

1. Create multiple boards that you want to run with a workload set on each board.
2. Create a simulator for each of the boards that you want to run. You can customize the simulator as well.
3. Add the simulator to the `multisim` object.
4. Execute all of the simulations that have been added to the `multisim` object.

Caveats:

- The output may be a bit messy if you use stdout.
- The output directories will be named after the `id` parameter of the simulator.
- There are still some rough edges.

Examples of using multisim

To add a board and simulator to multisim, use `multisim.add_simulator`. Do this once for each simulation you want to do.

```
multisim.add_simulator(
    Simulator(
        board=board,
        id="my_simulator"
    )
)
```

Using the multisim execution module

You can use the `multisim` module to run the simulations, list them, or run one simulation.

Run all simulations:

```
gem5 -m gem5.utils.multisim my_script.py
```

List all simulations:

```
gem5 my_script.py --list
```

Run a single simulation:

```
gem5 my_script.py {id}
```

Exercise: Use suites and workloads

In this exercise, we will first simply print all of the workloads in a suite.

Then we will run one of the workloads from the suite.

Finally, we will use `multisim` to run all of the workloads in parallel and compare the results for different workloads and configurations.

We will use the `BigProcessor` and `LittleProcessor` classes from the previous exercise.

Questions

1. What is the IPC of each workload on the big and little processors?
2. How much speedup did we achieve by using `multisim`?

Step 1: Create the configurations

Let's go back and copy the Big and Little processor configurations from the previous exercise. Modify the `BigProcessor` and `LittleProcessor` classes to have a `get_name` **class method** that returns the name of the processor.

This will allow us to easily identify the processor in the output.

Step 1: Answer

```
class BigProcessor(BaseCPUProcessor):
    def __init__(self):
        super().__init__(
            cores=[BigCore()]
        )

    @classmethod
    def get_name(cls):
        return "big"
```

```
class LittleProcessor(BaseCPUProcessor):
    def __init__(self):
        super().__init__(
            cores=[LittleCore()]
        )

    @classmethod
    def get_name(cls):
        return "little"
```

Step 2: Write the script to create the simulators

Use the "riscv-getting-started-benchmark-suite" to get a set of workloads to run.

Run all of these workloads on both the big and little processors.

For the ID of the simulation, you can use `f"{processor_type.get_name()}-{benchmark.get_id()}"`.

Remember, in this script you are just adding the simulators to the `multisim` object.

Use the `multisim.set_num_processes` method to set the number of processes to run in parallel to "2".

Important: Each simulator must have its own *instance* of the comonents. I.e., you cannot reuse the same instance of the processor for multiple simulators.

Step 2: Answer

```
multisim.set_num_processes(2)
for processor_type in [BigProcessor, LittleProcessor]:
    for benchmark in obtain_resource("riscv-getting-started-benchmark-suite"):
        board = SimpleBoard( clk_freq="3GHz",
            processor=processor_type(), memory=SingleChannelDDR4_2400("1GiB"),
            cache_hierarchy=PrivateL1CacheHierarchy(
                l1d_size="32KiB", l1i_size="32KiB"
            ),
        )
        board.set_workload(benchmark)
        simulator = Simulator(
            board=board, id=f"{processor_type.get_name()}-{benchmark.get_id()}"
        )
        multisim.add_simulator(simulator)
```

Step 3: Run the simulations

Run the simulations using the `multisim` module.

This may take a while as some of these workloads larger than what we have used so far.

Step 3: Answer

```
gem5 -m gem5.utils.multisim my-cores-run.py
```

Exercise: Questions

1. What is the IPC of each workload on the big and little processors?
2. How much speedup did we achieve by using `multisim`?

For now, test with `gem5 my-cores-run.py --list`

And `gem5 my-cores-run.py little-riscv-npb-is-size-s-run` should run in 1-2 minutes and have an output in `m5out/little-riscv-npb-is-size-s-run`.