gem5's Standard Library

Using gem5's python interface

Using gem5

Remember: gem5's interface is *Python*.

Computer systems are complex. A declarative interface (e.g., `ini` or `json`) files, would be difficult to use.

Assuming we want to write *code* to describe the system we are going to simulate, we don't want to have to write down every single detail every time.

Python is well-suited for these kinds of "domain-specific languages."

You can think of gem5 similar to TensorFlow or PyTorch. It's a *framework* or *language* for describing and simulating computer systems.

Why a Standard Library?

- Using stdlib you don't need to specify 1,000s-10,000s lines to describe the system.
- Standard "components" reduces
  - Duplicated code.
  - Error-prone configurations.
  - A lack of portability between different simulation setups.

Previously, there was `se.py` and `fs.py`

- These tried to be everything to everyone
- "Spaghetti code"
- The default "interface" to gem5 was massive bash lines and hacks

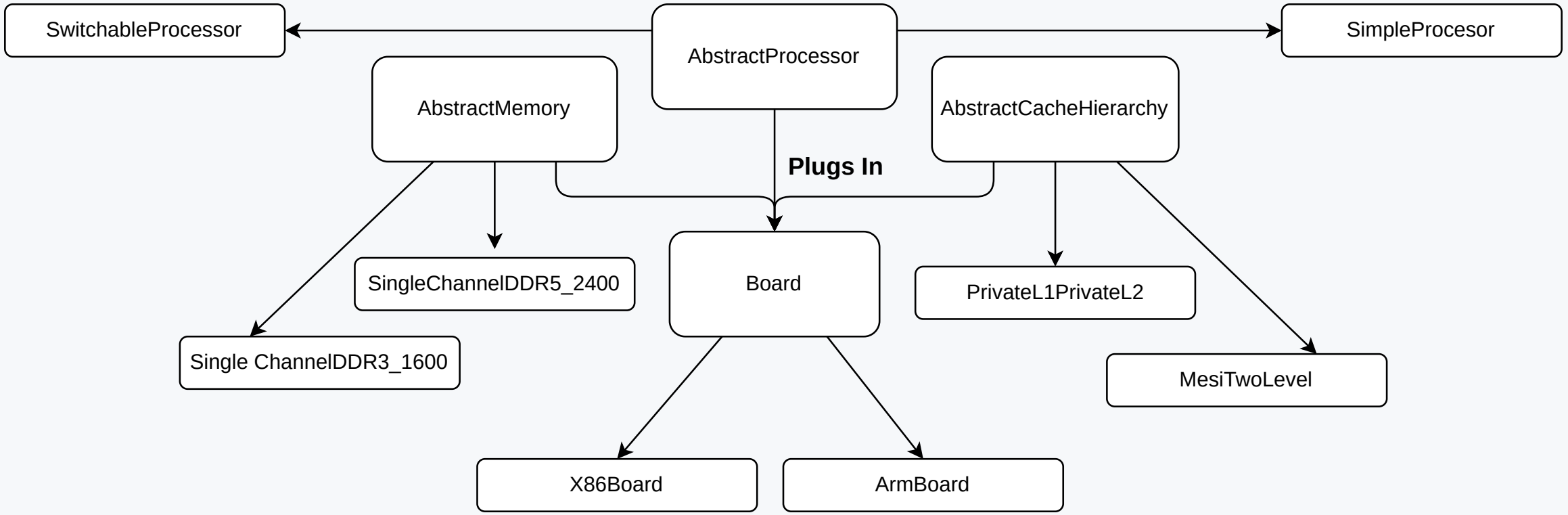
Think of gem5 more like TensorFlow than a command line tool. **gem5** is a *framework* or *language*.

What is the Standard Library (stdlib)?

The purpose of the gem5 Standard Library is to provide a set of predefined components that can be used to build a simulation that does the majority of the work for you.

For the remainder that is not supported by the standard library, APIs are provided that make it easy to extend the library for your own use.

The metaphor: Plugging components together into a board



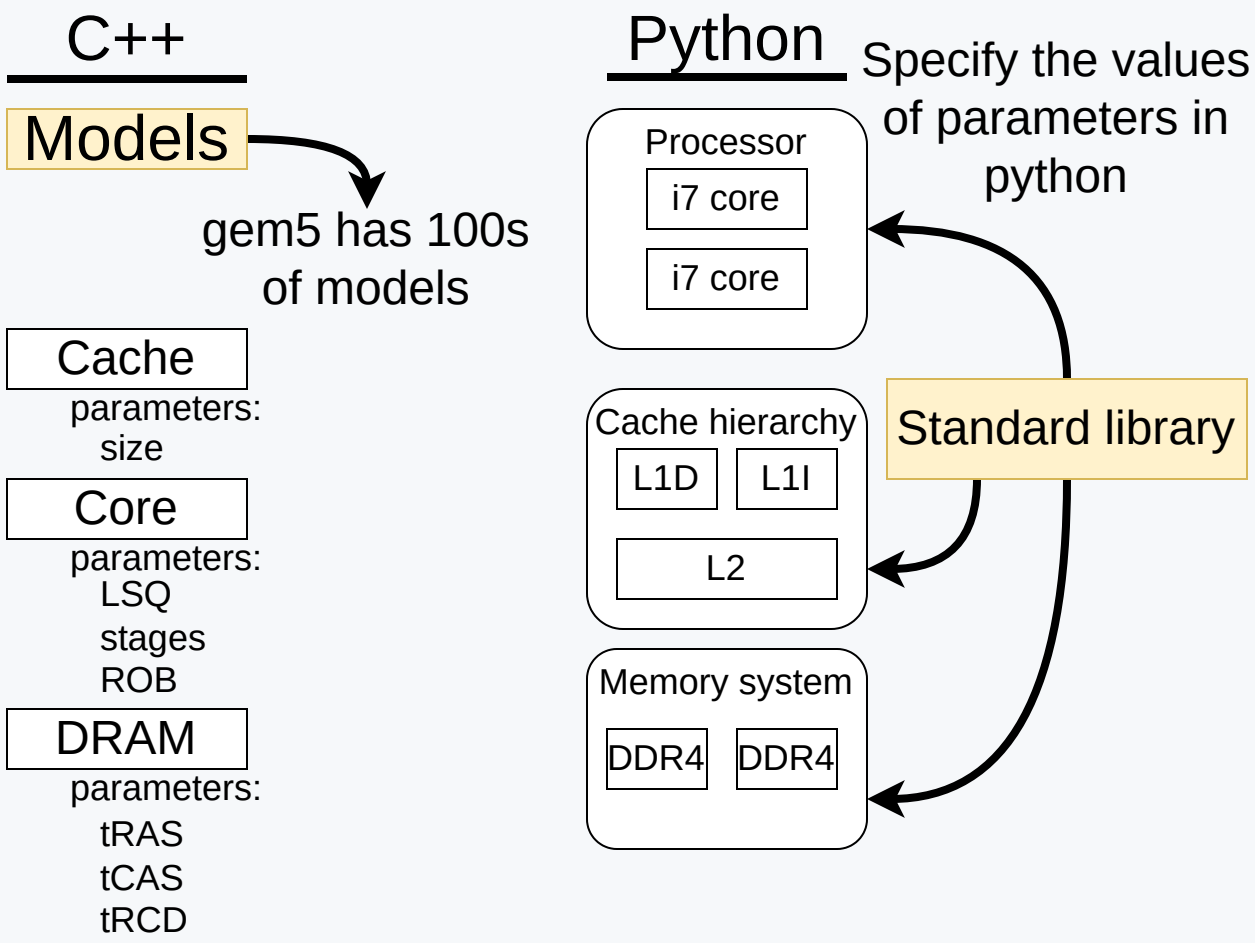
Main idea

Due to its modular, object-oriented design, gem5 can be thought of as a set of components that can be plugged together to form a simulation.

The types of components are *boards*, *processors*, *memory systems*, and *cache hierarchies*:

- **Board**: The "backbone" of the system. You plug components into the board. The board also contains the system-level things like devices, workload, etc. It's the boards job to negotiate the connections between other components.
- **Processor**: Processors connect to boards and have one or more *cores*.
- **Cache hierarchy**: A cache hierarchy is a set of caches that can be connected to a processor and memory system.
- **Memory system**: A memory system is a set of memory controllers and memory devices that can be connected to the cache hierarchy.

Remember: gem5 software architecture



Python

Control the simulation

```
sim.start()
sim.dump_stats()
sim.stop_at_inst(...)
```

Develop experiments with the key parameters

Exercise: Build your own board

Characteristics

- A single core Arm CPU using a "simple" model.
- Two-level cache hierarchy with 32 KiB 8-way L1 and 512 KiB 16-way L2.
- Single channel DDR4 2400 memory.

Run BFS from the GAP benchmark suite.

Questions

- What is the average IPC?
- What is the total simulated time?
- What is the output of the simulated program?

Step 0

Here are the imports you need to get started. It's a lot, but we'll see each one of them as we go through the bootcamp over the next few days.

```
from gem5.components.boards.simple_board import SimpleBoard
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.components.cachehierarchies.ruby.mesi_two_level_cache_hierarchy import (
    MESITwoLevelCacheHierarchy,
)
from gem5.components.memory.single_channel import SingleChannelDDR4_2400
from gem5.components.processors.cpu_types import CPUTypes
from gem5.isas import ISA
from gem5.resources.resource import obtain_resource
from gem5.simulate.simulator import Simulator
```

Step 1: Instantiate a processor

Use the `SimpleProcessor` class to create a simple processor that will run in timing mode, uses the Arm ISA, and has 1 core.

See `SimpleProcessor` for hints.

Step 1: Answer

```
processor = SimpleProcessor(cpu_type=CPUTypes.TIMING, isa=ISA.ARM, num_cores=1)
```

`SimpleProcessor` is a component that allows you to customize the model for the underlying cores.

The `cpu_type` parameter specifies the type of CPU model to use.
(We'll see more on this later.)

The `isa` parameter specifies the ISA that the CPU will execute.
gem5 supports many different ISAs.

The `num_cores` parameter specifies the number of cores in the processor.
The `SimpleProcessor` assumes homogenous cores.

Step 2: Instantiate a cache hierarchy

Use the `MESITwoLevelCacheHierarchy` for the cache hierarchy.

Use 32 KiB 8-way L1 caches, 256 KiB 16-way L2 caches, and a single L2 bank.

See `MESITwoLevelCacheHierarchy` for hints.

Step 2: Answer

```
cache_hierarchy = MESITwoLevelCacheHierarchy(
    l1d_size="32KiB",
    l1d_assoc=8,
    l1i_size="32KiB",
    l1i_assoc=8,
    l2_size="256KiB",
    l2_assoc=16,
    num_l2_banks=1,
)
```

`MESITwoLevelCacheHierarchy` is a component that represents a two-level MESI cache hierarchy. This uses the Ruby memory model.

The component for the cache hierarchy is parameterized with the sizes and associativities of the L1 and L2 caches.

Step 3: Instantiate a memory system

Use the `SingleChannelDDR4_2400` for the memory system.

See `SingleChannel` classes for hints.

Step 3: Answer

```
memory = SingleChannelDDR4_2400()
```

This component represents a single-channel DDR4 memory system.

There is a `size` parameter that can be used to specify the capacity of the memory of the simulated system. You can reduce the size to save simulation time, or use the default for the memory type (e.g., one channel of DDR4 defaults to 8 GiB).

There are also multi-channel memories available.
We'll cover this more in Memory Systems.

Step 4: Plug components into the board

Use the `SimpleBoard` class to create a board that will run the simulation.
Use a 3 GHz clock frequency.

See `SimpleBoard` for hints.

Step 4: Answer

A `SimpleBoard` is a board which can run any ISA in Syscall Emulation (SE) mode.
It is "Simple" due the relative simplicity of SE mode.
Most boards are tied to a specific ISA and require more complex designs to run Full System (FS) simulation.

```
board = SimpleBoard(
    clk_freq="3GHz",
    processor=processor,
    memory=memory,
    cache_hierarchy=cache_hierarchy,
)
```

Step 5: Set up the workload

Use the `set_workload` method of the board to set the workload.
See the `set_workload` function for hints.

Use the `obtain_resource` function to download the files needed to run the specified workload.

Use the "arm-gapbs-bfs-run" workload.

Step 5: Answer

```
board.set_workload(obtain_resource("arm-gapbs-bfs-run"))
```

The `obtain_resource` function downloads the files needed to run the specified workload. In this case "arm-gapbs-bfs-run" is a BFS workload from the GAP Benchmark Suite.

This uses "gem5 resources"

Here we can search the available resources: https://resources.gem5.org/.

Here is the arm-gabps-bfs-run resource: https://resources.gem5.org/resources/arm-gapbs-bfs-run?version=1.0.0.

Step 6: Set up the simulation and run

Use the `Simulator` class to set up the simulation and run it.
See the `Simulator` class for hints.

Use the `run` method to run the simulation.

Step 6: Answer

Set up the simulation:

```
simulator = Simulator(board=board)
simulator.run()
```

(More on this later, but this is object that controls the simulation loop).

Run it

```
gem5-mesi 01-components.py
```

Exercise questions

<iframe src="https://app.sli.do/event/qpr43XWrbjYJCdE3GHGCWg/embed/polls/63d7ea52-2bb4-45a8-ae01-9aa3ecf044cd" width="800" height="150"></iframe> <iframe src="https://app.sli.do/event/qpr43XWrbjYJCdE3GHGCWg/embed/polls/ae905f28-f1cd-475f-a758-a59be860157d" width="800" height="150"></iframe> <iframe src="https://app.sli.do/event/qpr43XWrbjYJCdE3GHGCWg/embed/polls/fe32093c-1c7a-4008-95d3-1010caa51057" width="800" height="150"></iframe>

Output

```
Generate Time:       0.00462
Build Time:          0.00142
Graph has 1024 nodes and 10496 undirected edges for degree: 10
Trial Time:          0.00010
Trial Time:          0.00008
Trial Time:          0.00008
Trial Time:          0.00008
Trial Time:          0.00008
Trial Time:          0.00009
Trial Time:          0.00008
Trial Time:          0.00008
Trial Time:          0.00008
Trial Time:          0.00011
Average Time:        0.00009
```

stats.txt

```
simSeconds                                   0.009093
simTicks                                   9093461436
```

Future things to consider

- How to change the processor?
  - The number of cores
  - The type of CPU model
  - Details of the CPU model (e.g., pipeline depth)
  - The ISA
- How to change the cache hierarchy?
  - The sizes and associativities of the L1 and L2 caches
  - The number of L2 banks
  - How to change the hierarchy (e.g., 3-level, 2-level, write-through)
- How to change the memory system?
  - The size of the memory
  - The type of memory (e.g., DDR3, DDR4, HBM)
  - The number of channels

We'll cover all of this in the coming sections.

Overview of stdlib components

A brief overview of the different kinds of components in the stdlib

Components included in gem5

```
gem5/src/python/gem5/components
----/boards
----/cachehierarchies
----/memory
----/processors

gem5/src/python/gem5/prebuilt
----/demo/x86_demo_board
----/riscvmatched
```

- gem5 stdlib in `src/python/gem5`
- Two types
  - Prebuilt: full systems with set parameters
  - Components: Components to build systems
- Prebuilt
  - Demo: Just examples to build off of
  - riscvmatched: Model of SiFive Unmatched

Components: Boards

```
gem5/src/python/gem5/components
----/boards
    ----/simple
    ----/arm_board
    ----/riscv_board
    ----/x86_board
----/cachehierarchies
----/memory
----/processors
```

- Boards: Things to plug into
  - Have "set_workload" and "connect_things"
- Simple: SE-only, configurable
- Arm, RISC-V, and X86 versions for full system simulation

Components: Cache hierarchies

```
gem5/src/python/gem5/components
----/boards
----/cachehierarchies
    ----/chi
    ----/classic
    ----/ruby
----/memory
----/processors
```

- Have fixed interface to processors and memory
- **Ruby:** detailed cache coherence and interconnect
- **CHI:** Arm CHI-based protocol implemented in Ruby
- **Classic caches:** Hierarchy of crossbars with inflexible coherence

A bit more about cache hierarchies

- Quick caveat: You need different gem5 binaries for different protocols
- Any binary can use classic caches
- Only one Ruby protocol per gem5 binary

In your codespaces, we have some pre-built binaries

- `gem5`: CHI (Fully configurable; based on Arm CHI)
- `gem5-mesi`: MESI_Two_Level (Private L1s, Shared L2)
- `gem5-vega`: GPU_VIPER (CPU: Private L1/L2 core pairs, shared L3; GPU: Private L1, shared L2)

Components: Memory systems

```
gem5/src/python/gem5/components
----/boards
----/cachehierarchies
----/memory
    ----/single_channel
    ----/multi_channel
    ----/dramsim
    ----/dramsys
    ----/hbm
----/processors
```

- Pre-configured (LP)DDR3/4/5 DIMMs
  - Single and multi channel
- Integration with DRAMSim and DRAMSys
  - Not needed for accuracy, but useful for comparisons
- HBM: An HBM stack

Components: Processors

```
gem5/src/python/gem5/components
----/boards
----/cachehierarchies
----/memory
----/processors
    ----/generators
    ----/simple
    ----/switchable
```

- Mostly "configurable" processors to build off of.
- Generators
  - Synthetic traffic, but act like processors.
  - Have linear, random, and more interesting patterns
- Simple
  - Only default parameters, one ISA.
- Switchable
  - We'll see this later, but you can switch from one to another during simulation.

More on processors

- Processors are made up of cores.
- Cores have a "BaseCPU" as a member. This is the actual CPU model.
- `Processor` is what interfaces with `CacheHierarchy` and `Board`
- Processors are organized, structured sets of cores. They define how cores connect with each other and with outside components and the board though standard interface.

gem5 has three (or four or five) different processor models

More details coming in the CPU Models section.

- `CPUTypes.TIMING`: A simple in-order CPU model
  - This is a "single cycle" CPU. Each instruction takes the time to fetch and executes immediately.
  - Memory operations take the latency of the memory system.
  - OK for doing memory-centric studies, but not good for most research.

CPU types

Other options for CPU types

- `CPUTypes.O3`: An out-of-order CPU model
  - Highly detailed model based on the Alpha 21264.
  - Has ROB, physical registers, LSQ, etc.
  - Don't use `SimpleProcessor` if you want to configure this.
- `CPUTypes.MINOR`: An in-order core model
  - A high-performance in-order core model.
  - Configurable four-stage pipeline
  - Don't use `SimpleProcessor` if you want to configure this.
- `CPUTypes.ATOMIC`: Used in "atomic" mode (more later)
- `CPUTypes.KVM`: More later

Summary

- gem5's standard library is a set of components that can be used to build a simulation that does the majority of the work for you.
- The standard library is designed around *extension* and *encapsulation*.
- The main types of components are *boards*, *processors*, *memory systems*, and *cache hierarchies*.
- The standard library is designed to be modular and object-oriented.
- The `Simulator` object controls the simulation.