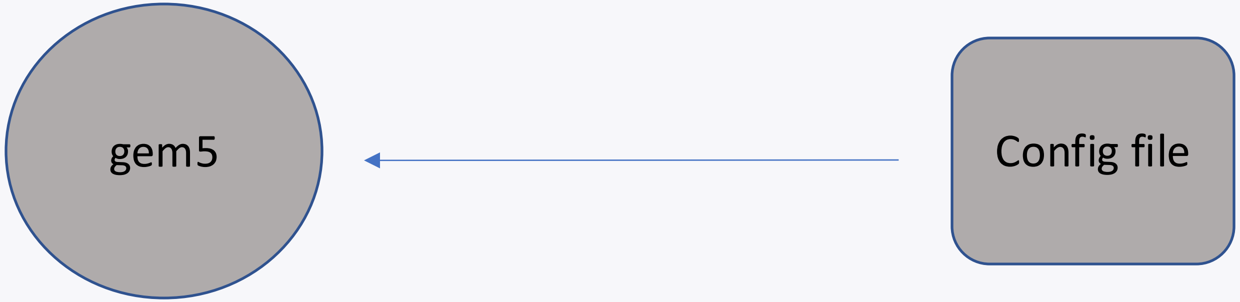
gem5 standard library

In this section, we will learn how to build a system to simulate using gem5's standard library, extend the standard library, use the simulator class to improve simulation performance, and use gem5 resources

Outline

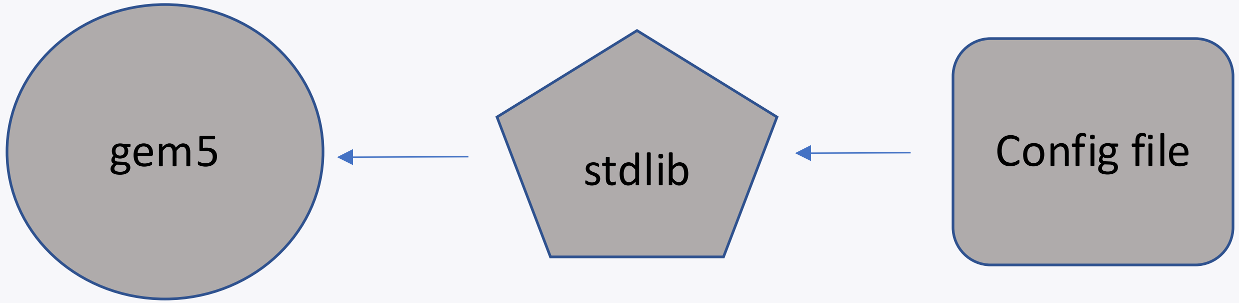
- Standard library concepts
- Using components module
- Using the simulator module
- KVM
- Checkpointing
- Using gem5 resources

Setting up simulations



This allows for maximum flexibility but can mean creating 100s of lines of Python to create even a basic simulation.

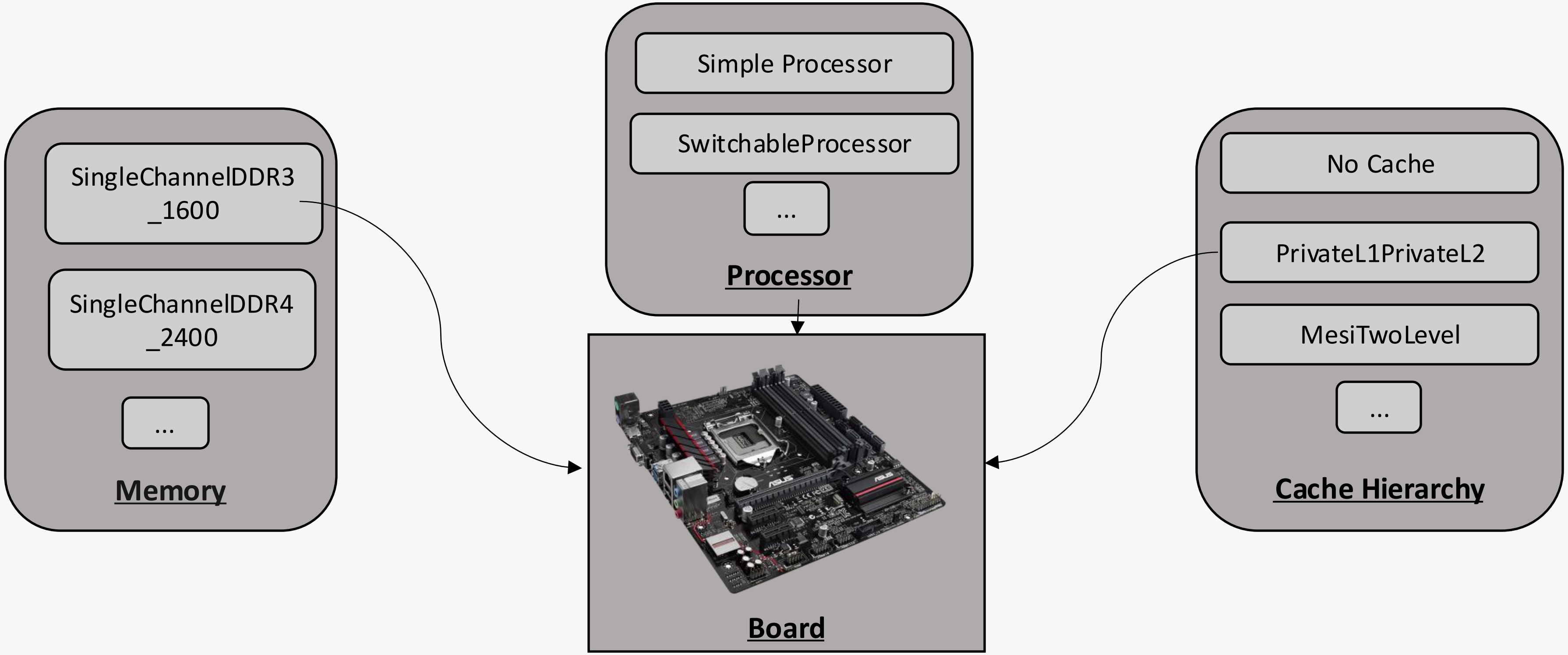
What is the standard library for?



The stdlib is a library which allows for users to quickly create systems with pre-built components.

The stdlib's module architecture allows for components (e.g. a memory system or a cache hierarchy setup) to be quickly swapped in and out without radical redesign.

The stdlib modular metaphor



gem5

Let’s build using components!

```
from gem5.components.boards.simple_board import SimpleBoard
from gem5.components.cachehierarchies.classic.private_l1_shared_l2_cache_hierarchy import (
    PrivateL1SharedL2CacheHierarchy,
)
from gem5.components.memory.single_channel import SingleChannelDDR4_2400
from gem5.components.processors.cpu_types import CPUTypes
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.isas import ISA
from gem5.resources.resource import obtain_resource
from gem5.simulate.simulator import Simulator
```

Open “materials/02-components.py” You’ll see the above already prepared for you.

Choose a cache, memory, & processor

```
main_memory = SingleChannelDDR4_2400(size="2GB")
caches = PrivateL1SharedL2CacheHierarchy(
    l1d_size="32KiB",
    l1d_assoc=8,
    l1i_size="32KiB",
    l1i_assoc=8,
    l2_size="256KiB",
    l2_assoc=16,
)
simple_in_order_core = SimpleProcessor(
    cpu_type=CPUTypes.TIMING, num_cores=1, isa=ISA.X86
)
```

Quick note on `SimpleProcessor`

- This does not model any particular processor
- It’s simple to allow you to get started
- Don’t use this for research if you care about CPU performance

Then let’s plug them into a board

```python
board = SimpleBoard(
    processor=simple_in_order_core,
    memory=main_memory,
    cache_hierarchy=caches,
    clk_freq="3GHz",
)
```

Load a workload into the board

```
board.set_workload(obtain_resource("x86-npb-is-size-s-run"))
```

Search gem5 resources for more workloads

Run the simulator

```
> gem5 materials/02-components.py
```

A completed version of the configuration can be found in “materials/isca24/completed/02-components.py”

To run

```
> gem5 materials/02-components.py
```

This will take about 40 seconds

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: Models 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

Components: 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
  - 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

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

- Preconfigured (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
- Simple
  - Only default parameters, one ISA
- Switchable
  - We'll see this later, but you can switch from one to another

gem5

Components: Processors

- Processors are made up of “cores”
- Cores have a “BaseCPU” as a member
- Processor is what interfaces with CacheHierarchy and Board

Components: Processors

- gem5 has three (or four or five) different processor models
- O3CPU:
  - Out-of-order CPU with ROB, BP, Physical register file, etc.
  - Highly configurable, but not very accurate to modern cores
- MinorCPU:
  - In-order CPU with 4 stage pipeline, BP, etc.
  - Highly configurable, similar to modern high-performance in-order designs
- SimpleCPU:
  - TimingSimpleCPU: Every instruction takes 0 cycles (just fetch time) except memory
  - AtomicSimpleCPU: Used in atomic mode (more later)
- KVMCPU: more later

Components: Processors

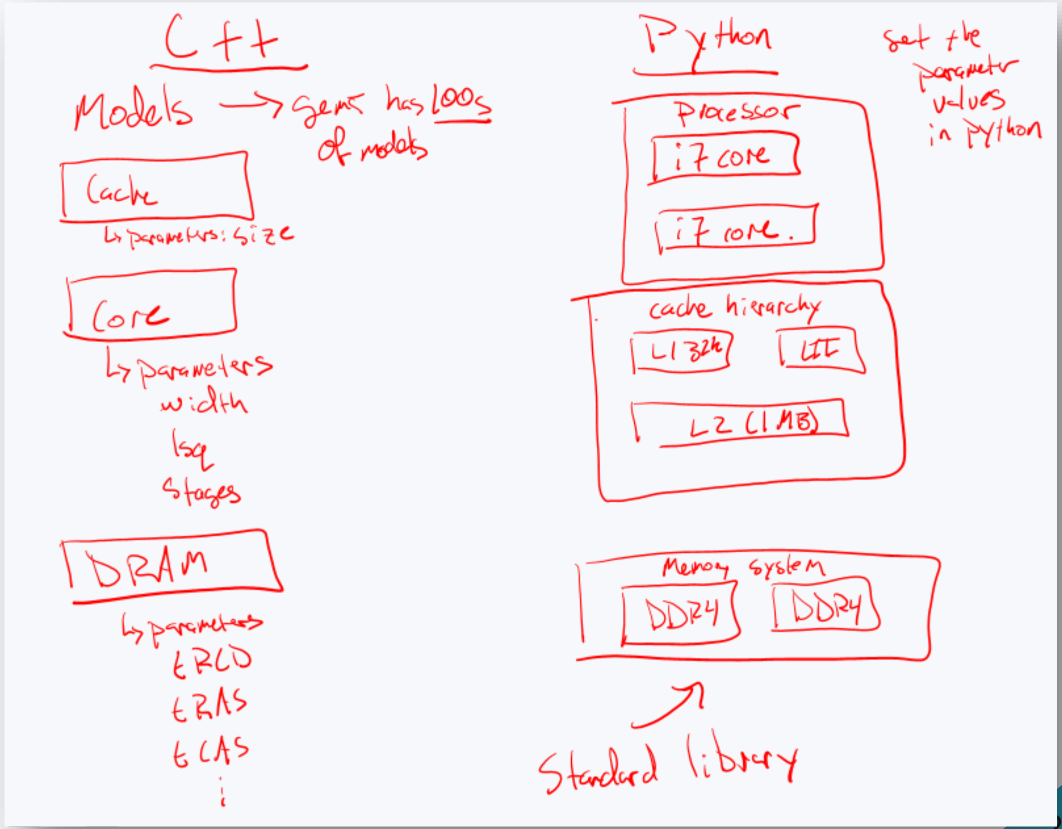
- O3CPU and MinorCPU are highly configurable
- See “BaseO3CPU.py” and “BaseMinorCPU.py”
- Each instructions type can have its own functional unit
  - Or one functional unit for many instruction types
  - Functional units can specify the latency and pipeline latency
- Many other options as well

Standard Library components

- Designed around *Extension* and *Encapsulation*
  - NOT designed for “parameterization”
- If you want to create a processor/cache hierarchy/etc. with different parameters
  - *Extend* using object-oriented semantics
- Let’s see and example

Quick reminder of gem5's architecture

- We will now create a new *component*
- Specialize/extend the "BaseO3CPU" (core)



Let’s create a processor with OOO cores

- Use materials/isca24/03-processor.py
- Mostly the same as before, but now

```
my_ooo_processor = MyOutOfOrderProcessor(
    width=8, rob_size=192, num_int_regs=256, num_fp_regs=256
)
```

Create subclass of BaseCPUProcessor/Core

```python
from m5.objects import X86O3CPU
from m5.objects import TournamentBP

class MyOutOfOrderCore(BaseCPUCore):
    def __init__(self,
        width, rob_size, num_int_regs,
        num_fp_regs):
        super().__init__(X86O3CPU(), ISA.X86)
```

- See src/python/gem5/components/processors/base_cpu_core.py
- And src/cpu/o3/BaseO3CPU.py

```python
        self.core.fetchWidth = width
        self.core.decodeWidth = width
        self.core.renameWidth = width
        self.core.issueWidth = width
        self.core.wbWidth = width
        self.core.commitWidth = width

        self.core.numROBEntries = rob_size

        self.core.numPhysIntRegs = num_int_regs
        self.core.numPhysFloatRegs = num_fp_regs

        self.core.branchPred = TournamentBP()

        self.core.LQEntries = 128
        self.core.SQEntries = 128
```

Create subclass of BaseCPUProcessor/Core

- The CPUProcessor assumes a list of cores that are BaseCPUCores

```
class MyOutOfOrderProcessor(BaseCPUProcessor):
    def __init__(self, width, rob_size, num_int_regs, num_fp_regs):
        cores = [MyOutOfOrderCore(width, rob_size, num_int_regs, num_fp_regs)]
        super().__init__(cores)
```

Now, run it and compare!

```
> gem5 materials/03-processor.py
```

- Takes 2-3 minutes
- Faster than simple in order?
- Use `--outdir=m5out/ooo` and `--outdir=simple`
- Compare the stats.txt (which stat?)

Controlling the simulation

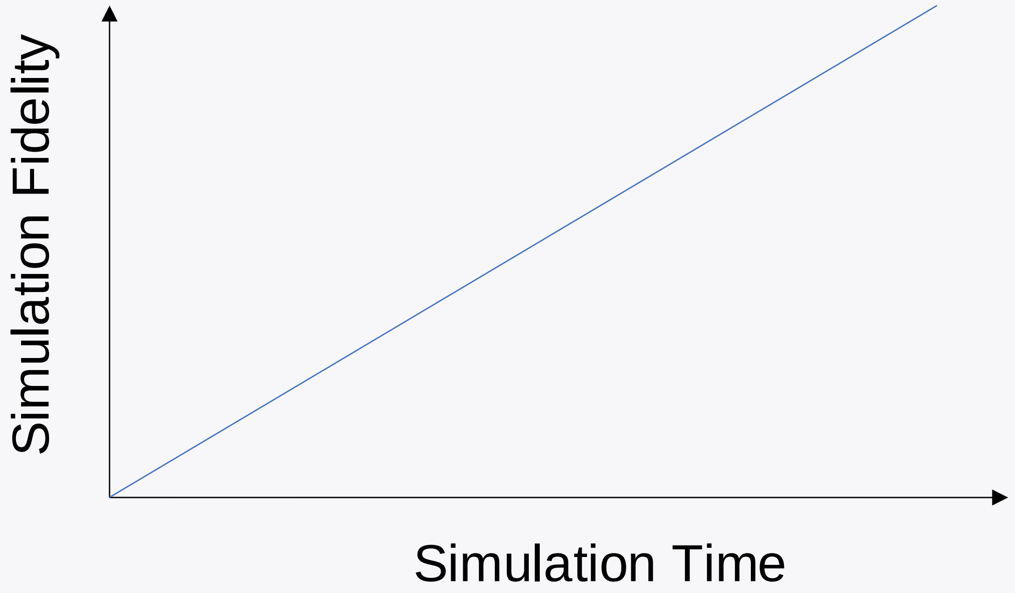
Let’s shift gears a bit and talk about controlling the simulation, improving performance, marking regions of interest, fast forwarding, and more

gem5 is sllooww

(Not our fault: It’s the natural of simulation)

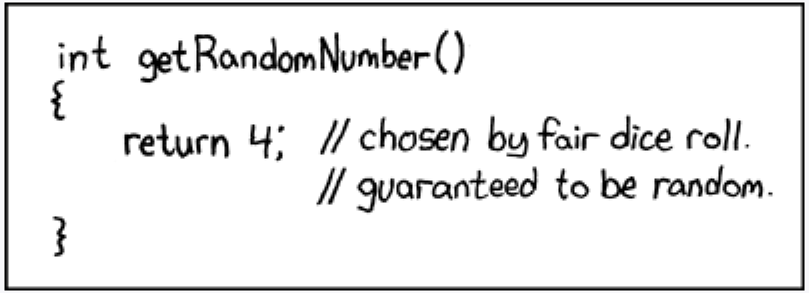
$$\begin{tikzcd}\text{Simulating 1 Second} \arrow[r] & \text{gem5} \arrow[r] & \text{>> 100k seconds on the host}\end{tikzcd}$$

Fortunately, there are some work arounds



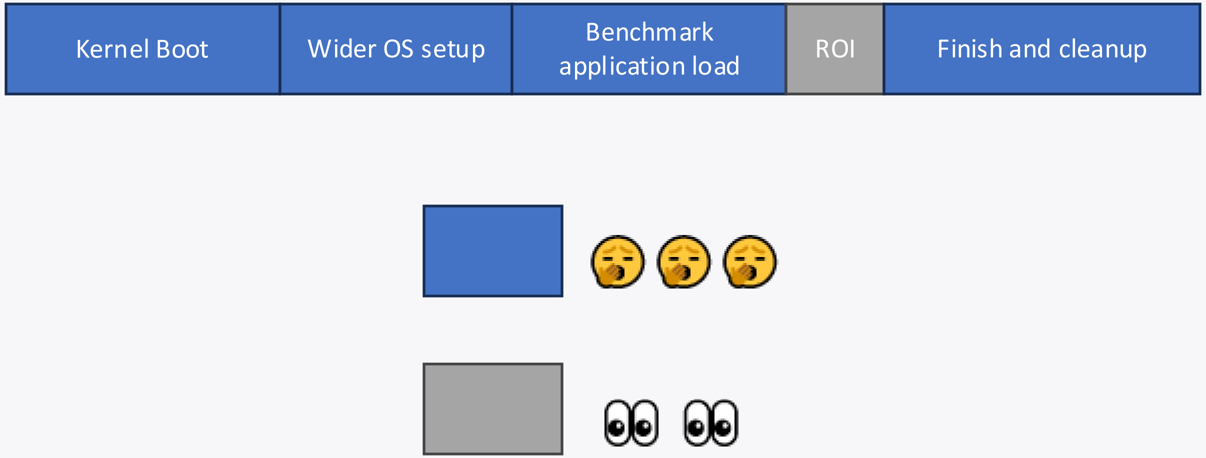
Key idea: You don't need to simulate everything perfectly, or at all.

Simulations can always be made faster by simulating less



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

This isn't always a bad thing… a lot of a simulation is of no interest to us



Some techniques we provide

CPU Models

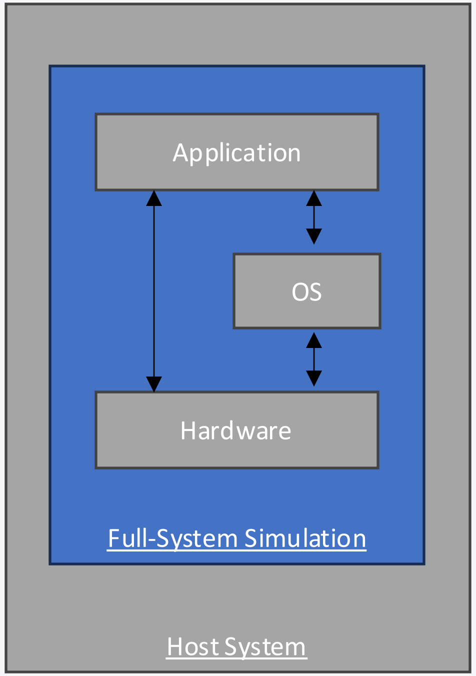
KVM Mode

SE Mode

Checkpoint

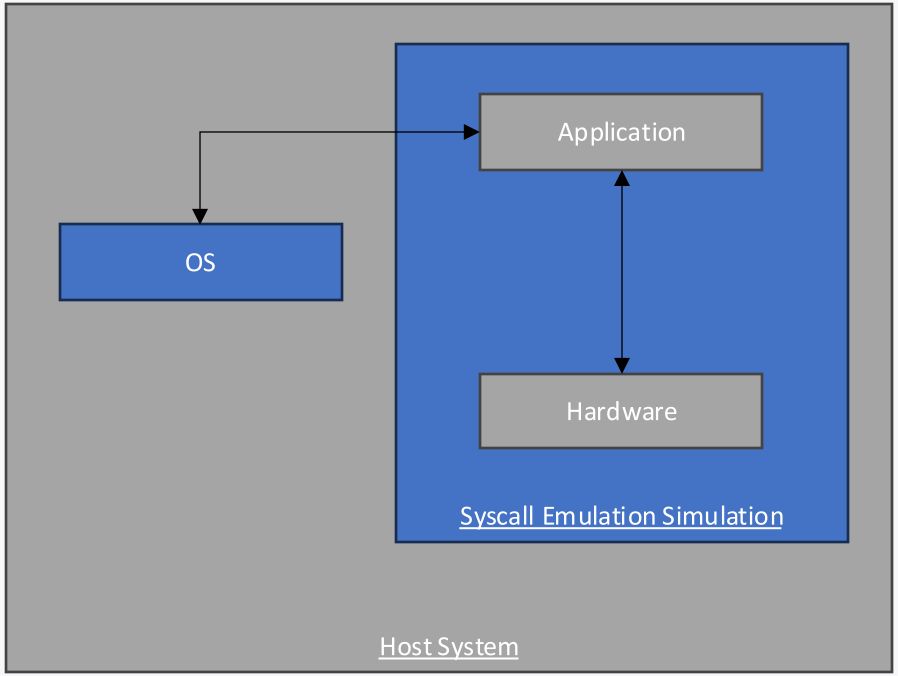
Sampling: Simpoints and Loopoints

SE mode vs FS mode

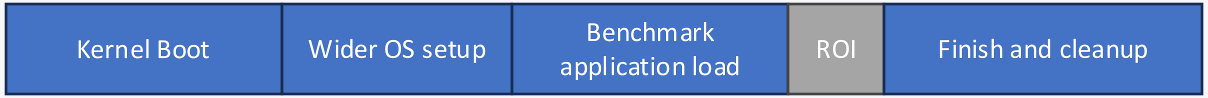


SE Mode relays application syscalls to host OS. This means we don't need to simulate an OS for applications to run

In addition, we can access host resources such as files of libraries to dynamically link in.



Goal: just run ROI in detailed mode



- Two ways:
  - Use KVM CPU to fast forward
  - Take a checkpoint

Fast-forwarding with KVM

- KVM: Kernel-based virtual machine
- Uses hardware virtualization extensions (e.g., nested page tables, vmexit, etc.)
- gem5 uses KVM as the “CPU model”
- I.e., the code is actually executing on the host CPU

- It is **fast!**
- But, your host must match your guest

Checkpointing

- Saves the architectural state of the system
- Some microarchitectural state is saved
- Depends on the models you're using in the simulation
- Can be reused many times
- Only some parts of the system can change between checkpoint and restore

KVM

- Very fast (nearly native speed)
- Flexible to simulation system changes
- Flexible to software changes

- Non-deterministic
- Host must match guest
- No RISC-V support
- Only CPU

Checkpointing

- Create once, run many times
- Almost all devices/components supported

- Cannot change software at all
- Only some simulation system changes
- Significant disk space

How to designate ROI

- gem5 has “magic instructions” or “bridge calls” so the *guest* can communicate with the *simulator.* (Like hypercalls)
- Often called “m5ops” though we are trying to rename to “gem5-bridge”

```
#ifdef HOOKS
        roi_begin_();
#endif
/*   This is the main iteration */
    for( iteration=1; iteration<=MAX_ITERATION
    {
        if( CLASS != 'S' ) printf( "        %d
        rank( iteration );
    }
/*   End of timing, obtain maximum time of all 
    timer_stop( 0 );
    timecounter = timer_read( 0 );
#ifdef HOOKS
        roi_end_();
#endif
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void roi_begin_(){
    printf(" -------------------- ROI BEGIN --
    m5_work_begin(0,0);
}
```

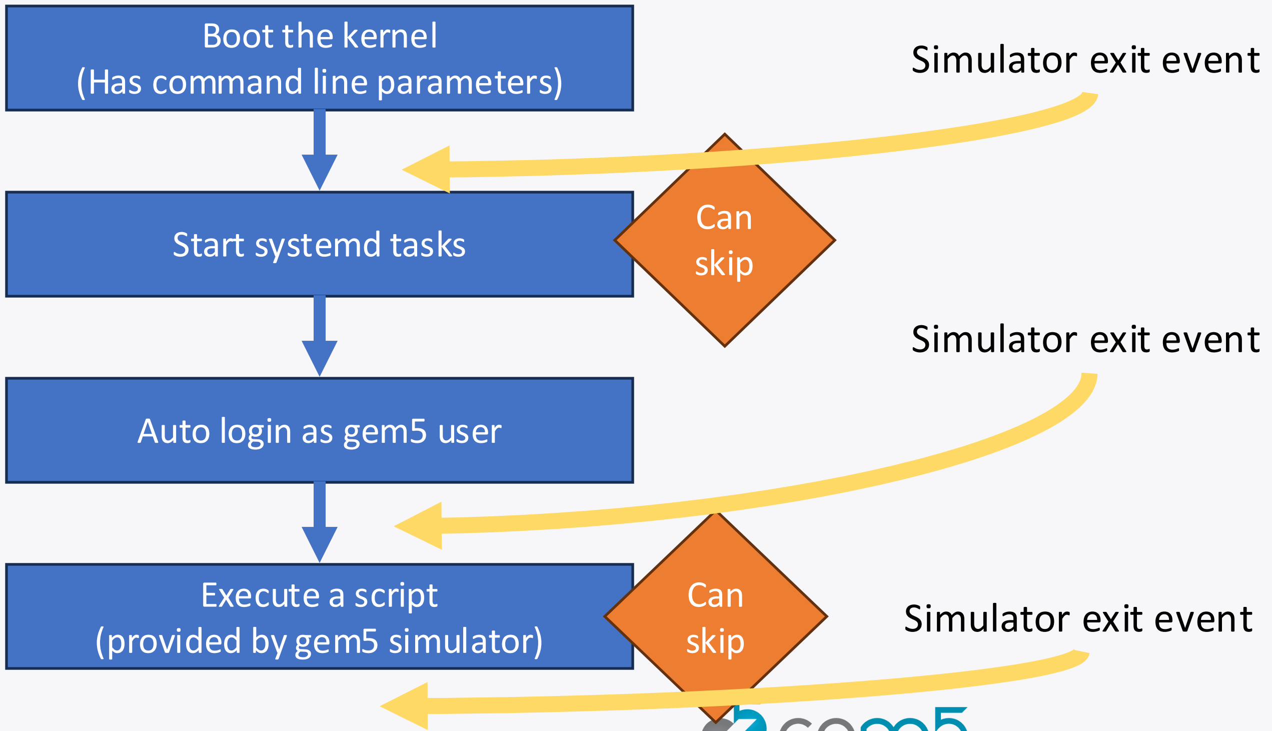
Let’s explore fast forwarding, checkpointing and ROI annotations

- Boot Ubuntu 24.04

```
workload = obtain_resource("x86-ubuntu-24.04-boot-with-systemd")
```

- Use materials/isca24/05-fs-npb.py

Booting Linux is complicated…



Controlling simulator events

```
def on_exit():
    print("Exiting the simulation for kernel boot")
    yield False
    print("Exiting the simulation for systemd complete")
    yield False
```

`on_exit` is a python “generator”
False means “keep going”
True means “exit gem5”

```
simulator = Simulator(
    board=board,
    on_exit_event={
        ExitEvent.EXIT: on_exit(),
    },
)
```

Many different kinds of exit events

- We use “EXIT” too much (changes coming soon)
- WORKBEGIN and WORKEND also common

```
    EXIT = "exit"  # A standard vanilla exit.
    WORKBEGIN = "workbegin"  # An exit because a ROI has been reached.
    WORKEND = "workend"  # An exit because a ROI has ended.
    SPATTER_EXIT = "spatter exit"  # An exit because a spatter core has ended.
    SWITCHCPU = "switchcpu"  # An exit needed to switch CPU cores.
    FAIL = "fail"  # An exit because the simulation has failed.
    CHECKPOINT = "checkpoint"  # An exit to load a checkpoint.
    SCHEDULED_TICK = "scheduled tick exit"
    MAX_TICK = "max tick"  # An exit due to a maximum tick value being met.
    USER_INTERRUPT = (  # An exit due to a user interrupt (e.g., cntr + c)
        "user interupt"
    )
    SIMPOINT_BEGIN = "simpoint begins"
    MAX_INSTS = "number of instructions reached"
    PERF_COUNTER_ENABLE = "performance counter enabled"
    PERF_COUNTER_DISABLE = "performance counter disabled"
    PERF_COUNTER_RESET = "performance counter reset"
```

How to interact?

- m5term (custom telnet)

```
> cd gem5/util/term
> make
```

Run the simulation

```
> gem5-mesi materials/04-kvm.py
```

- In another window, watch it

```
> ./m5term 3456
```

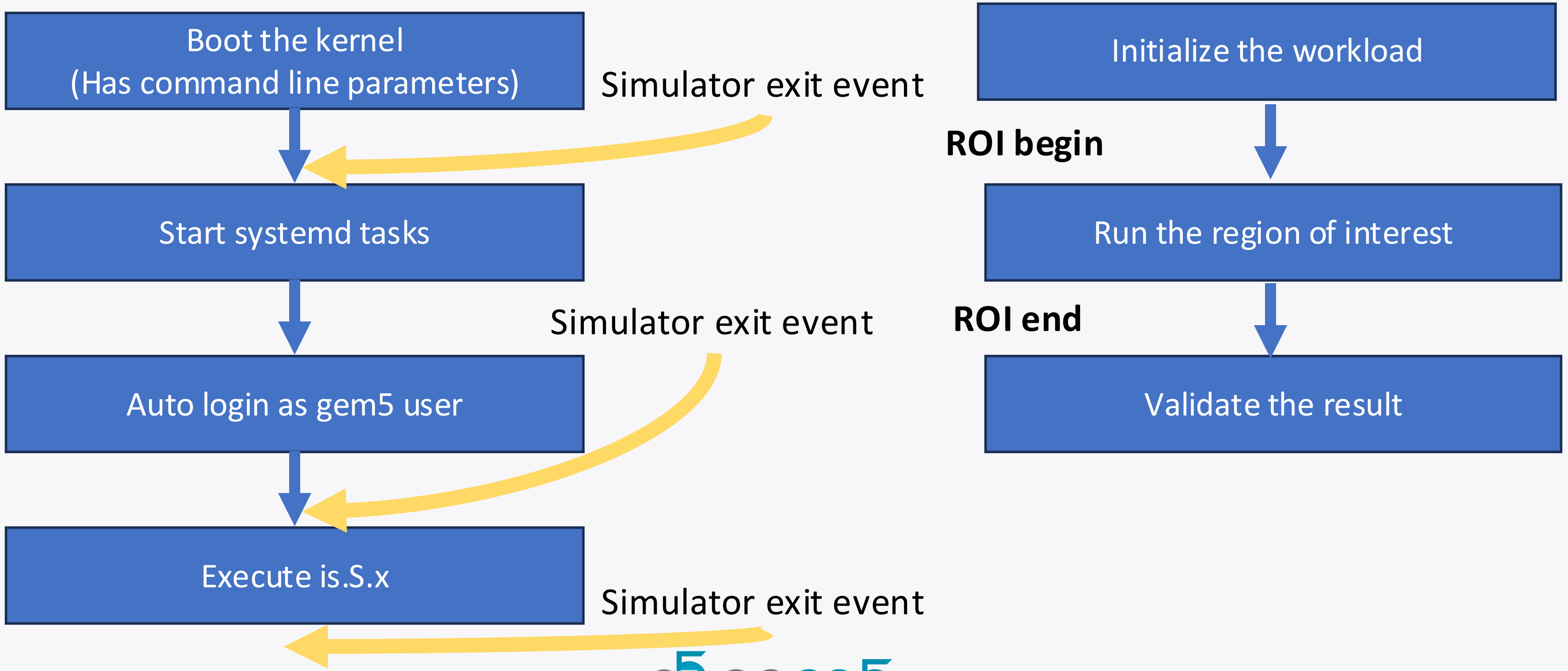
Next step, let's switch cores at ROI

```
workload = obtain_resource("x86-ubuntu-24.04-npb-is-s-run")
```

- What does this do?
- https://resources.gem5.org/resources/x86-npb-is-size-s

- Boots ubuntu, executes `/home/gem5/NPB3.4-OMP/bin/is.S.x`

Running workloads is complicated…



gem5

Fast-forward with KVM, detailed TIMING

- Use 05-fs-npb.py
- Mostly the same as 04-kvm.py

```
def on_work_begin():
    print("Work begin")
    m5.stats.reset()
    processor.switch()
    yield False

def on_work_end():
    print("Work end")
    yield True
```

```
simulator = Simulator(
    board=board,
    on_exit_event={
        ExitEvent.EXIT: on_exit(),
        ExitEvent.WORKBEGIN: on_work_begin(),
        ExitEvent.WORKEND: on_work_end(),
    },
)
```

Look at stats.txt

- Output only for the ROI!
- Overall, very fast

- Please use FS mode ☺

Checkpointing

- As easy as `m5.checkpoint("checkpoint")`
- Let’s checkpoint after booting Linux, but before the workload
- See 06-npb-checkpoint.py

```
def on_exit():
    print("Exiting the simulation for kernel boot")
    yield False
    print("Exiting the simulation for systemd complete")
    m5.checkpoint("checkpoint")
    yield True
```

Note:
SwitchableProcessor isn’t good for checkpointing

gem5

Checkpointing

```
> gem5-mesi materials/05-npb-checkpoint.py
```

Restoring

- 06-npb-restore.py

```
processor = SimpleSwitchableProcessor(
    starting_core_type=CPUTypes.KVM,
    switch_core_type=CPUTypes.O3,
    isa=ISA.X86,
    num_cores=1,
)
```

```
board.set_kernel_disk_workload(
    kernel=obtain_resource("x86-linux-kernel-5.4.0-105-generic"),
    disk_image=obtain_resource("x86-ubuntu-24.04-npb-img"),
    kernel_args=[
            "earlyprintk=ttyS0",
            "console=ttyS0",
            "lpj=7999923",
            "root=/dev/sda2"
        ],
    readfile_contents=f"echo 12345 | sudo -S /home/gem5/NPB3.4-OMP/bin/sp.S.x; sleep 5;
m5 exit;"
    checkpoint=CheckpointResource("checkpoint")
)
```

Restoring

- Note: We can’t always change the workload like this
- We took the checkpoint before the file was read in.
- Checkpoints are *separate* workloads from their non-checkpoint counterparts (in gem5-resources too!)

Restoring

```
def on_work_begin():
    print("Work begin.")
    m5.stats.reset()
    processor.switch()
    yield False
```

```
def on_work_end():
    print("Work end")
    yield False

simulator = Simulator(
    board=board,
    on_exit_event={
        ExitEvent.WORKEND: on_work_end(),
    },
)
```

Running the restore

- sp takes longer ~5 minutes with timing &

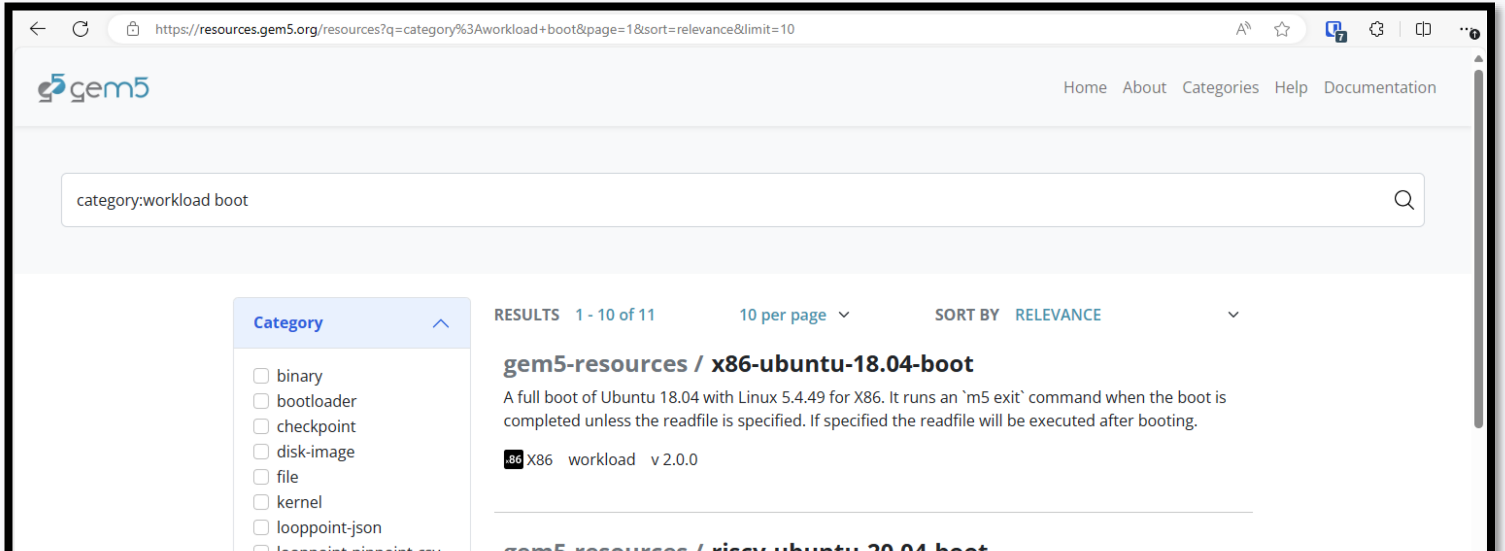
```
> gem5-mesi materials/06-npb-restore.py
```

```
> tail m5out/board.pc.com_1.device # or m5term 3456
```

gem5 resources

gem5 resources

- Setting up simulations is complicated
  - Kernel, operating system, libraries, workloads, annotations, inputs, etc.
- gem5 resources contains everything you need


https://resources.gem5.org/resources?q=category%3Aworkload+boot&page=1&sort=relevance&limit=10

category:workload boot

RESULTS 1 - 10 of 11 10 per page SORT BY RELEVANCE

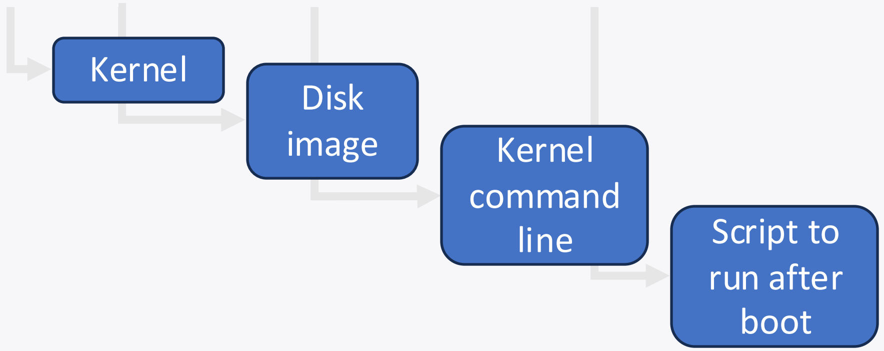
gem5-resources / x86-ubuntu-18.04-boot

A full boot of Ubuntu 18.04 with Linux 5.4.49 for X86. It runs an `m5 exit` command when the boot is completed unless the readfile is specified. If specified the readfile will be executed after booting.

X86 workload v 2.0.0

gem5 resources

```
board = X86Board()
board.set_workload(obtain_resource("x86-ubuntu-22.04-boot-with-systemd"))
simulator = Simulator(board=board)
simulator.run()
```



Types of resources

- Files: Binaries, Kernels, Disk images, Bootloaders
- Directories: Checkpoint, Simpoint
- Workload: Combination of other resources and options
- Suite: Set of workloads

Using resources in boards

- `board.set_workload(obtain_resource(...))`
  - Sets the workload with the “set workload function” specified in workload
- Can also set SE/FS workloads directly
- Uses “mixins” on the board class (slightly confusing)
- See src/python/gem5/components/boards/*_workload.py

board.set_se_*_workload

```
def set_se_binary_workload(
        self,
        binary: BinaryResource,
        exit_on_work_items: bool = True,
        stdin_file: Optional[FileResource] = None,
        stdout_file: Optional[Path] = None,
        stderr_file: Optional[Path] = None,
        env_list: Optional[List[str]] = None,
        arguments: List[str] = [],
        checkpoint: Optional[Union[Path, CheckpointResource]] = None,
    ) -> None:
        """Set up the system to run a specific binary.
```

board.set_kernel_disk_workload

```
    def set_kernel_disk_workload(
        self,
        kernel: KernelResource,
        disk_image: DiskImageResource,
        bootloader: Optional[BootloaderResource] = None,
        disk_device: Optional[str] = None,
        readfile: Optional[str] = None,
        readfile_contents: Optional[str] = None,
        kernel_args: Optional[List[str]] = None,
        exit_on_work_items: bool = True,
        checkpoint: Optional[Union[Path, CheckpointResource]] = None,
    ) -> None:
        """
        This function allows the setting of a full-system run with a Kernel
        and a disk image.
```

Using local resources

- File/directory resources can be created with just a path

```
BinaryResource('./hello.exe')
```

Creating local resource databases

- Create a file "gem5-config.json" in the current path
  - Can configure multiple different databases
- To use a json file
  - See https://www.gem5.org/documentation/gem5-stdlib/using-local-resources
- Use “Raw” tab on resource.gem5.org for help

```
[
    {
        "category": "binary",
        "id": "test-binary",
        "description": "A test binary",
        "architecture": "RISCV",
        "size": 1,
        "tags": [
            "test"
        ],
        "is_zipped": false,
        "md5sum": "6d9494d22b90d817e826b0d762fda973",
        "source": "src/simple",
        "url": "file:// path to fake_binary",
        "license": "",
        "author": [],
        "source_url": "https://github.com/gem5/gem5-resourc
        "resource_version": "1.0.0",
        "gem5_versions": [
            "23.0"
        ],
        "example_usage": "obtain_resource(resource_id=\"tes
    }
]
```

Workloads and suites

- Suites are a set of workloads
- Can iterate over all workloads
- Can filter based on “input group”

multisim

- Often, you want to run multiple workloads, “suites,” or design-space exploration
- multisim allows you to do this in parallel!
- Use materials/isca24/07-multisim.py

```python
for workload in obtain_resource("riscv-getting-started-benchmark-suite"):
    board.set_workload(workload)
    simulator = Simulator(board=board, id=workload.get_id())

    add_simulator(simulator)
```

multisim

- A couple of weird things
- 1. The script you write declares a set of “simulators” to run
  - The simulator has the board, workload, etc.
- 2. You don’t run it with just “gem5” you have to use a special module

```
> gem5 -m gem5.utils.multisim 07-multisim.py
```

multisim

- List the simulations it will run

```
> gem5 07-multisim.py -l
```

- Run one simulation

```
> gem5 07-multisim.py riscv-npb-is-size-s-run
```

multisim

- More improvements coming!
  - Managing stdout from gem5 is a bit of a pain (can use -re, but still not great)
- Maybe a dashboard?
  - Time left estimate, check on status of simulations, etc.

LUNCH!