Running Things on gem5

What we will cover

- Intro to Syscall Emulation mode
- m5ops
- Annotating workloads
- Cross-compiling workloads
- Traffic generator

Intro to Syscall Emulation Mode

What is Syscall Emulation mode, and when to use/avoid it

Syscall Emulation (SE) mode does not model all the devices in a system. It focuses on simulating the CPU and memory system. It only emulates Linux system calls, and only models user-mode code.

SE mode is a good choice when the experiment does not need to model the OS (such as page table walks), does not need a high fidelity model (emulation is ok), and faster simulation speed is needed.

However, if the experiment needs to model the OS interaction, or needs to model a system in high fidelity, then we should use the full-system (FS) mode. The FS mode will be covered in 07-full-system.

Example

00-SE-hello-world

Under `materials/02-Using-gem5/03-running-in-gem5/00-SE-hello-world`, there is a small example of an SE simulation.
00-SE-hello-world.py will run 00-SE-hello-world binary with a simply X86 configuration.
This binary does a print of the string `Hello, Worlds!`.
If we use the debug flag `SyscallAll` with it, we will able to see what syscalls are simulated.
We can do it with the following command:

```
gem5 -re --debug-flags=SyscallAll 00-SE-hello-world.py
```

00-SE-hello-world

Then in the simout.txt, we should see:

```
280945000: board.processor.cores.core: T0 : syscall Calling write(1, 21152, 14)...
Hello, World!
280945000: board.processor.cores.core: T0 : syscall Returned 14.
```

On the left, it is the timestamp for the simulation.
As the timestamp suggests, **SE simulation DOES NOT record the time for the syscall**.

m5ops

What is m5ops

- The **m5ops** (short for m5 opcodes) provide different functionalities that can be used to communicate between the simulated workload and the simulator.
- The commonly used functionalities are below. More can be found in the m5ops documentation:
  - exit [delay]: Stop the simulation in delay nanoseconds
  - workbegin: Cause an exit event of type "workbegin" that can be used to mark the begining of an ROI
  - workend: Cause and exit event of type "workend" that can be used to mark the ending of an ROI
  - resetstats [delay[period]]: Reset simulation statistics in delay nanoseconds; repeat this every period nanoseconds
  - dumpstats [delay[period]]: Save simulation statistics to a file in delay nanoseconds; repeat this every period nanoseconds
  - checkpoint [delay [period]]: Create a checkpoint in delay nanoseconds; repeat this every period nanoseconds
  - switchcpu: Cause an exit event of type, “switch cpu,” allowing the Python to switch to a different CPU model if desired

IMPORTANT

- ***Not all of the ops do what they say automatically***
- Most of these just only exit the simulation
- For example:
  - exit: Actually exits
  - workbegin: Only exits, if configured in `System`
  - workend: Only exits, if configured in `System`
  - resetstats: Resets the stats
  - dumpstats: Dumps the stats
  - checkpoint: Only exits
  - switchcpu: Only exits
- See gem5/src/sim/pseudo_inst.cc for details
- The gem5 standard library might have default behaviors for some of the m5ops. See src/python/gem5/simulate/simulator.py for the default behaviors

More about m5ops

There are three versions of m5ops:

1. Instruction mode: it only works with native CPU models
2. Address mode: it works with native CPU models and KVM CPU (only supports Arm and X86)
3. Semihosting: it works with native CPU models and Fast Model

Different modes should be used depending on the CPU type and ISA.

The address mode m5ops will be covered in 07-full-system as gem5-bridge and 08-accelerating-simulation after the KVM CPU is introduced.
In this session, we will only cover the instruction mode.

When to use m5ops

There are two main ways of using the m5ops:

1. Annotating workloads
2. Making gem5-bridge calls in disk images

In this session, we will focus on learning how to use the m5ops to annotate workloads.

How to use m5ops

m5ops provides a library of functions for different functionalities. All functions can be found in gem5/include/gem5/m5ops.h.
The commonly used functions (they are matched with the commonly used functionailites above):

- `void m5_exit(uint64_t ns_delay)`
- `void m5_work_begin(uint64_t workid, uint64_t threadid)`
- `void m5_work_end(uint64_t workid, uint64_t threadid)`
- `void m5_reset_stats(uint64_t ns_delay, uint64_t ns_period)`
- `void m5_dump_stats(uint64_t ns_delay, uint64_t ns_period)`
- `void m5_checkpoint(uint64_t ns_delay, uint64_t ns_period)`
- `void m5_switch_cpu(void)`

In order to call these functions in the workload, we will need to link the m5ops library to the workload. So first, we need to build the m5ops library.

Building the m5ops library

The m5 utility is in gem5/util/m5 directory.
In order to build the m5ops library,

1. `cd` into the `gem5/util/m5` directory
2. run `scons [{TARGET_ISA}.CROSS_COMPILE={TARGET_ISA CROSS COMPILER}] build/{TARGET_ISA}/out/m5`
3. the compiled library (`m5` is for command line utility, and `libm5.a` is a C library) will be at `gem5/util/m5/build/{TARGET_ISA}/out`

Notes

- If the host system ISA does not match with the target ISA, then we will need to use the cross-compiler
- `TARGET_ISA` has to be in lower case

Hands-on Time!

01-build-m5ops-library

Let's build the m5ops library for x86 and arm64

```
cd /workspaces/2024/gem5/util/m5
scons build/x86/out/m5
scons arm64.CROSS_COMPILE=aarch64-linux-gnu- build/arm64/out/m5
```

Linking the m5ops library to C/C++ code

After building the m5ops library, we can link them to our workload by:

1. Include **gem5/m5ops.h** in the workload's source file(s) (`<gem5/m5ops.h>`)
2. Add **gem5/include** to the compiler's include search path (`-Igem5/include`)
3. Add **gem5/util/m5/build/{TARGET_ISA}/out** to the linker search path (`-Lgem5/util/m5/build/{TARGET_ISA}/out`)
4. Link against **libm5.a** with (`-lm5`)

Hands-on Time!

02-annotate-this

Let's annotate the workload with `m5_work_begin` and `m5_work_end`

In `materials/02-Using-gem5/03-running-in-gem5/02-annotate-this`, there is a workload source file 02-annotate-this.cpp and a Makefile.

The workload mainly does two things:

1. Write a string to the standard out

```
write(1, "This will be output to standard out\n", 36);
```

02-annotate-this

2. Output all the file and folder names in the current directory

```
struct dirent *d;
DIR *dr;
dr = opendir(".");
if (dr!=NULL) {
    std::cout<<"List of Files & Folders:\n";
    for (d=readdir(dr); d!=NULL; d=readdir(dr)) {
        std::cout<<d->d_name<< ", ";
    }
    closedir(dr);
}
else {
    std::cout<<"\nError Occurred!";
}
std::cout<<std::endl;
```

02-annotate-this

Our goal in this exercise

- Mark `write(1, "This will be output to standard out\n", 36);` as our region of interest so we can see the execution trace of the syscall.

How do we do that?

1. Include the m5ops header file with `#include <gem5/m5ops.h>`
2. Call `m5_work_begin(0, 0);` right before `write(1, "This will be output to standard out\n", 36);`.
3. Call `m5_work_end(0, 0);` right after `write(1, "This will be output to standard out\n", 36);`
4. Compile the workload with the following requirments
    1. Add **gem5/include** to the compiler's include search path
    2. Add **gem5/util/m5/build/x86/out** to the linker search path
    3. Link against **libm5.a**

02-annotate-this

For step 4, we can modifiy the Makefile to have it run

```
$(GXX) -o 02-annotate-this 02-annotate-this.cpp \
  -I$(GEM5_PATH)/include \
  -L$(GEM5_PATH)/util/m5/build/$(ISA)/out \
  -lm5
```

If you are having any troubles, the completed version of everything is under `materials/02-Using-gem5/03-running-in-gem5/02-annotate-this/complete`.

02-annotate-this

If the workload is successfully compiled, we can try to run it with

```
./02-annotate-this
```

However, we will see the following error:

```
Illegal instruction (core dumped)
```

This is because the host does not recognize the instruction version of m5ops.
This is also the reason why we will need to use the address version of m5ops if we use the KVM CPU for our simulation.

Hands-on Time!

03-run-x86-SE

Let's write a handler to handle the m5 exit events

First, let's see what the default behavior is. Go to the folder `materials/02-Using-gem5/03-running-in-gem5/03-run-x86-SE` and run 03-run-x86-SE.py with the following command:

```
gem5 -re 03-run-x86-SE.py
```

After running the simulation, we should see a directory called `m5out` in `materials/02-Using-gem5/03-running-in-gem5/03-run-x86-SE`. Open the file `simerr.txt` in `m5out`. We should see two lines that look like this:

```
warn: No behavior was set by the user for work begin. Default behavior is resetting the stats and continuing.

warn: No behavior was set by the user for work end. Default behavior is dumping the stats and continuing.
```

03-run-x86-SE

As mentioned before, the gem5 standard library might have default behaviors for some of the m5ops. In here, we can see that it has default behaviors for `m5_work_begin` and `m5_work_end`.
Let's detour a bit to see how the gem5 standard library recognize the exit event and assign it a default exit event.
All standard library defined exit events can be found in src/python/gem5/simulate/exit_event.py. It uses the exit string of exit events to categories exit events. For example, both `"workbegin"` and `"m5_workend instruction encountered"` exit strings are categorized as `ExitEvent.WORKBEGIN`.
All pre-defined exit event handler can be found in src/python/gem5/simulate/exit_event_generators.py.

For example, the `ExitEvent.WORKBEGIN` defaults to use the `reset_stats_generator`. It means that when we are using the standard library `Simulator` object, if there is an exit with exit string `"workbegin"` or `"m5_workbegin instruction encountered"`, it will automatically execute `m5.stats.reset()` unless we over-write the default behavior using the `on_exit_event` parameter in the gem5 stdlib `Simulator` parameter.

03-run-x86-SE

Let's add custom workbegin and workend handlers, and use the `on_exit_event` parameter in `Simulator` parameter to over-write the default behaviors. To do this, add the following into 03-run-x86-SE.py:

```
# define a workbegin handler
def workbegin_handler():
    print("Workbegin handler")
    m5.debug.flags["ExecAll"].enable()
    yield False
#
# define a workend handler
def workend_handler():
    m5.debug.flags["ExecAll"].disable()
    yield False
#
```

Also, register the handlers using the `on_exit_event` parameter in the `Simulator` object construction

```
# setup handler for ExitEvent.WORKBEGIN and ExitEvent.WORKEND
    on_exit_event= {
        ExitEvent.WORKBEGIN: workbegin_handler(),
        ExitEvent.WORKEND: workend_handler()
    }
#
```

03-run-x86-SE

Let's run this simulation again with the following command

```
gem5 -re 03-run-x86-SE.py
```

Now, we will see the following in [materials/02-Using-gem5/03-running-in-gem5/03-run-x86-SE/m5out/simout.txt]

```
3757178000: board.processor.cores.core: A0 T0 : 0x7ffff7c82572 @_end+140737350460442    :   syscall                  : IntAlu :   flags=()
This will be output to standard out
3757180000: board.processor.cores.core: A0 T0 : 0x7ffff7c82574 @_end+140737350460444    : cmp   rax, 0xfffffffffffff000
```

This shows the log of the debug flag `ExecAll` that we enabled for our ROI using the `m5.debug.flags["ExecAll"].enable()`, it shows all the execution trace for our ROI. As the timestamp on the left suggested again, SE mode **DOES NOT** time the emulated system calls. Also, as the log suggested, we over-wrote the default behavior of the `m5_work_begin` and `m5_work_end`.

Then, with the output

```
List of Files & Folders:
., .., 03-run-SE.py, m5out,
Simulation Done
```

it indicates that SE mode is able to read files on the host machine. Additionally, SE mode is able to write files on the host machine.

However, again, SE mode is **NOT** able to time the emulated system calls.

Tips on SE mode

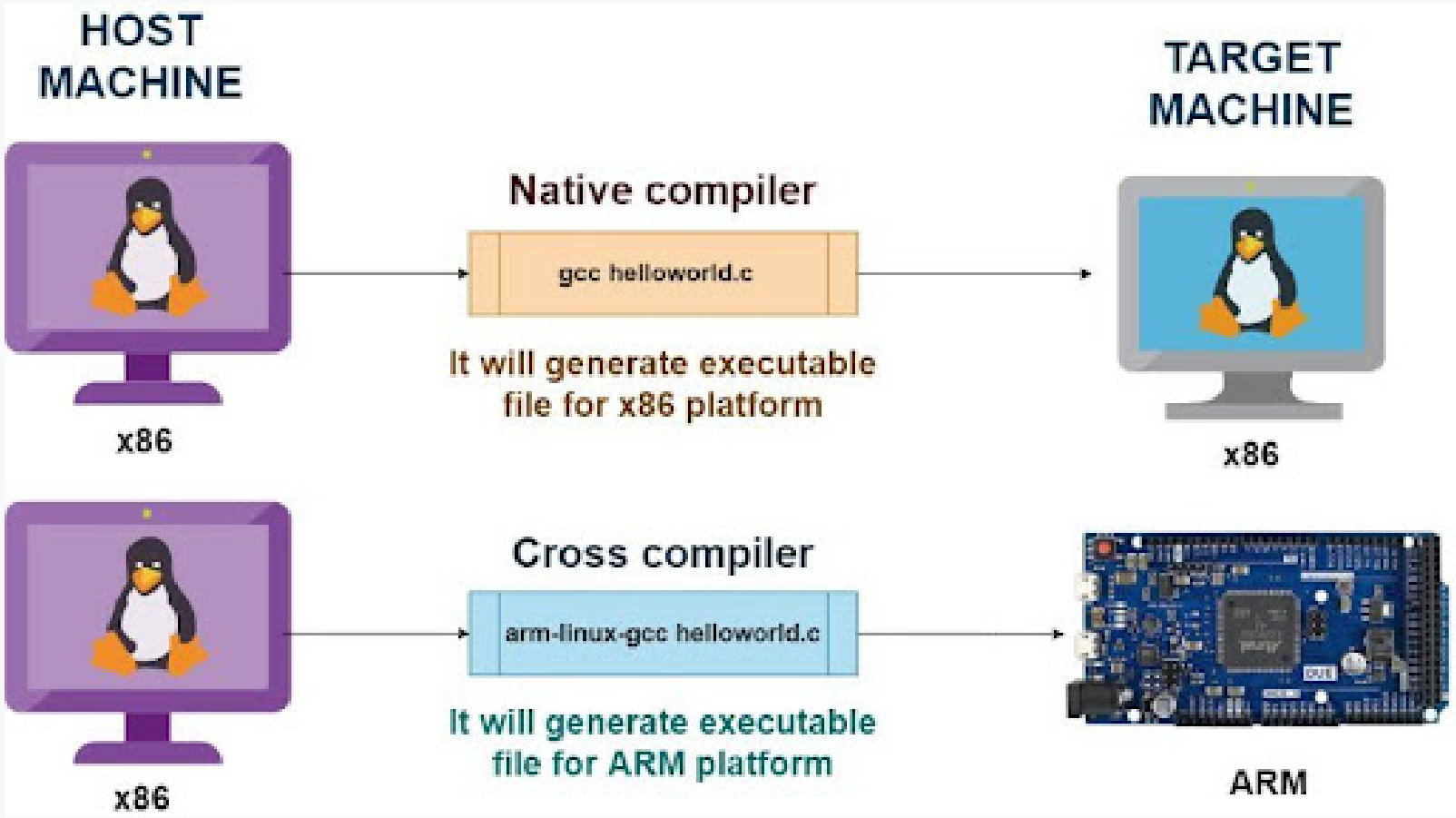
With the gem5 stdlib, we usually use the `set_se_binary_workload` function in the `board` object to setup the workloads. We can pass in files, arguments, enviornment variables, and output file paths to the `set_se_binary_workload` function using the corresponding parameters.

```
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:
```

For more information, we can look at src/python/gem5/components/boards/se_binary_workload.p.

Cross-compiling

Cross-compiling from one ISA to another.



Hands-on Time!

04-cross-compile-workload

Let's cross compile the workload to arm64 statically and dynamically

For static compilation, add the following command to the Makefile in `materials/02-Using-gem5/03-running-in-gem5/04-cross-compile-workload`:

```
$(GXX) -o 04-cross-compile-this-static 04-cross-compile-this.cpp -static -I$(GEM5_PATH)/include -L$(GEM5_PATH)/util/m5/build/$(ISA)/out -lm5
```

For dynamic compilation, add the following command:

```
$(GXX) -o 04-cross-compile-this-dynamic 04-cross-compile-this.cpp -I$(GEM5_PATH)/include -L$(GEM5_PATH)/util/m5/build/$(ISA)/out -lm5
```

Next, run `make` in the same directory as the Makefile.

04-cross-compile-workload

Notes:

Note that we are using `arm64` as the ISA and `aarch64-linux-gnu-g++` for the cross compiler. This is in contrast to exercise 2, where the ISA was `x86` and the compiler was `g++`.

Also note that the the static compilation command has the flag `-static`, while the dynamic command has no additional flags.

Hands-on Time!

05-run-arm-SE

Let's run the compiled arm64 workloads and see what happens

First, let's run the statically compiled workload. `cd` into the directory `materials/02-Using-gem5/03-running-in-gem5/05-run-arm-SE` and run `05-run-arm-SE.py` using the following command:

```
gem5 -re --outdir=static 05-run-arm-SE.py --workload-type=static
```

Next, let's run the dynamically compiled workload with the following command:

```
gem5 -re --outdir=dynamic 05-run-arm-SE.py --workload-type=dynamic
```

05-run-arm-SE

You will see the following error output in `dynamic/simout.txt` from running the dynamically compiled workload:

```
src/base/loader/image_file_data.cc:105: fatal: fatal condition fd < 0 occurred: Failed to open file /lib/ld-linux-aarch64.so.1.
This error typically occurs when the file path specified is incorrect.
Memory Usage: 217652 KBytes
```

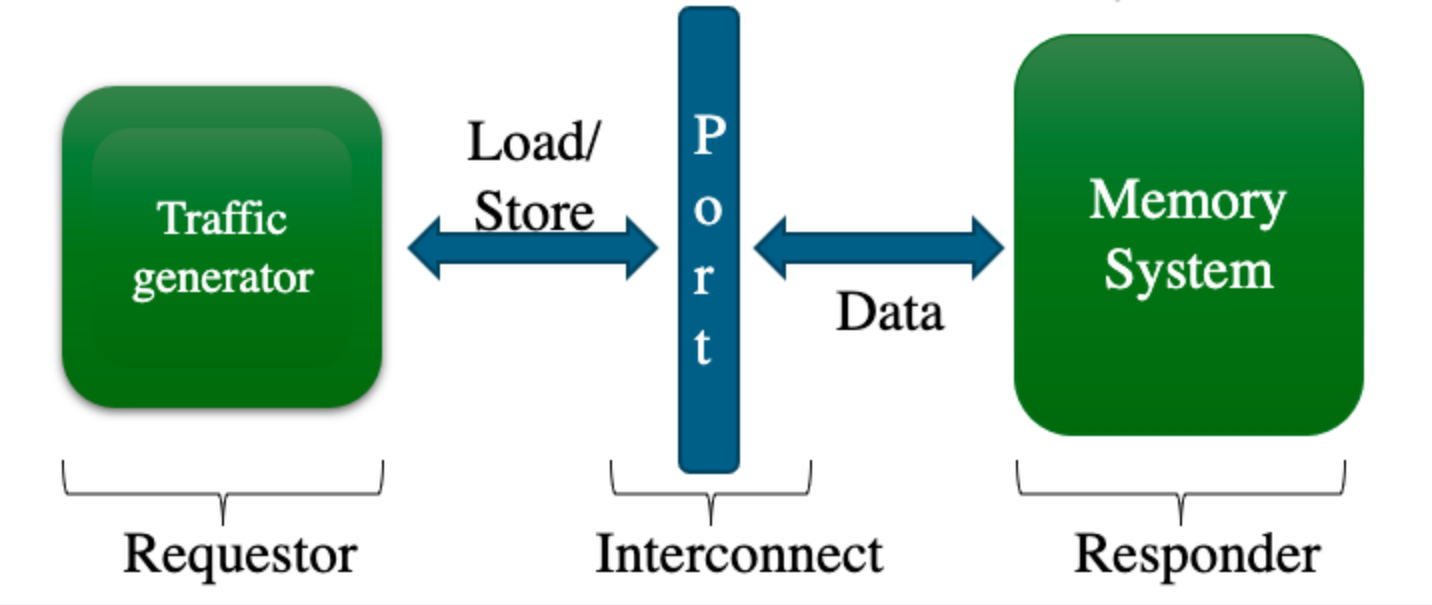
To use the dynamically compiled workload, we will have to redirect the library path. We can do this by adding the following to the configuration script, under `print("Time to redirect the library path")`:

```
setInterpDir("/usr/aarch64-linux-gnu/")
board.redirect_paths = [RedirectPath(app_path=f"/lib",
                        host_paths=[f"/usr/aarch64-linux-gnu/lib"])]
```

Traffic Generator in gem5

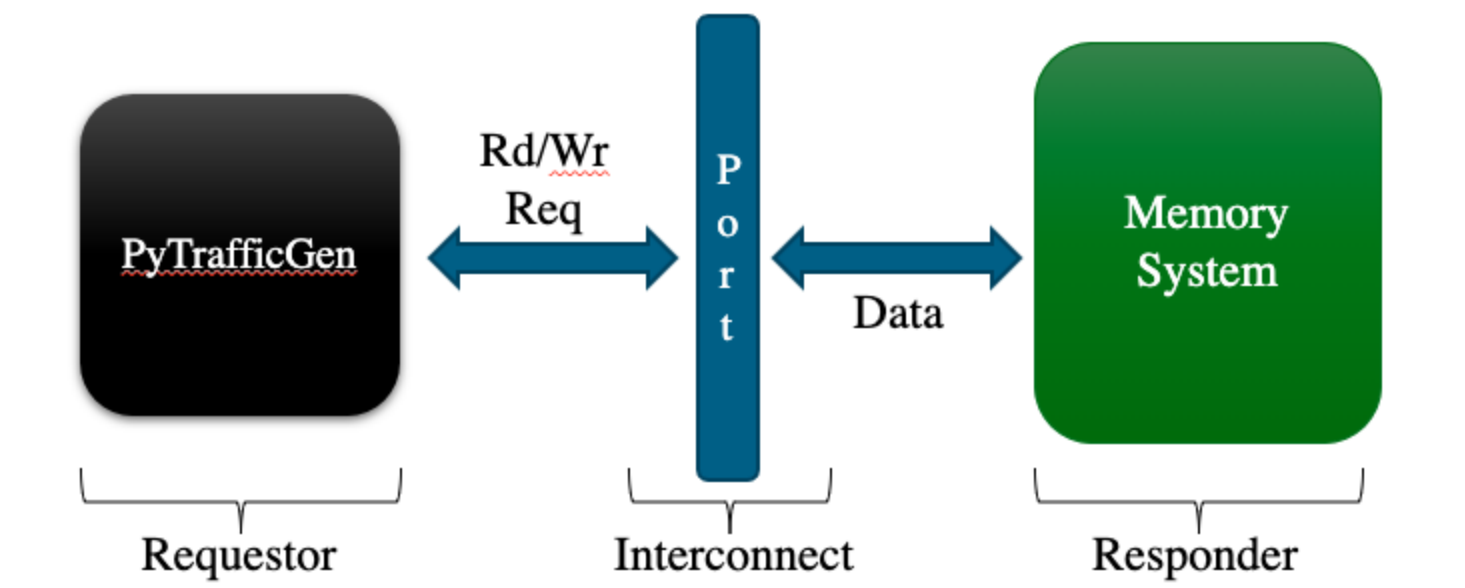
Traffic Generator

- A traffic generator module generates stimuli for the memory system.
- Used for creating test cases for caches, interconnects, and memory controllers, etc.



gem5's Traffic Gen: PyTrafficGen

- PyTrafficGen is a traffic generator module (SimObject) located in: `gem5/src/cpu/testers/traffic_gen`
- Used as a black box replacement for any generator of read/write requestor.



PyTrafficGen: Params

- PyTrafficGen’s parameters allow you to control the characteristics of the generated traffic.

Parameter	Definition
pattern	The pattern of generated addresses: linear/ random
duration	The duration of generating requests in ticks (quantum of time in gem5).
start address	The lower bound for addresses that the synthetic traffic will access.
end address	The upper bound for addresses that the synthetic traffic will access.
minimum period	The minimum timing difference between two consecutive requests in ticks.
maximum period	The maximum timing difference between two consecutive requests in ticks.
request size	The number of bytes that are read/written by each request.
read percentage	The percentage of reads among all the requests, the rest of requests are write requests.

Hands-on Time!

06-traffic-gen

Let's run an example on how to use the traffic generator

Summery

SE mode does NOT implement many things!

- Filesystem
- Most of systemcalls
- I/O devices
- Interrupts
- TLB misses
- Page table walks
- Context switches
- multiple threads
  - You may have a multithreaded execution, but there's no context switches & no spin locks

More summaries

m5ops can be used to communicate between simulated workload and the simulator

Traffic generator can abstract away the details of a data requestor such as CPU for generating test cases for memory systems