Modeling caches in gem5

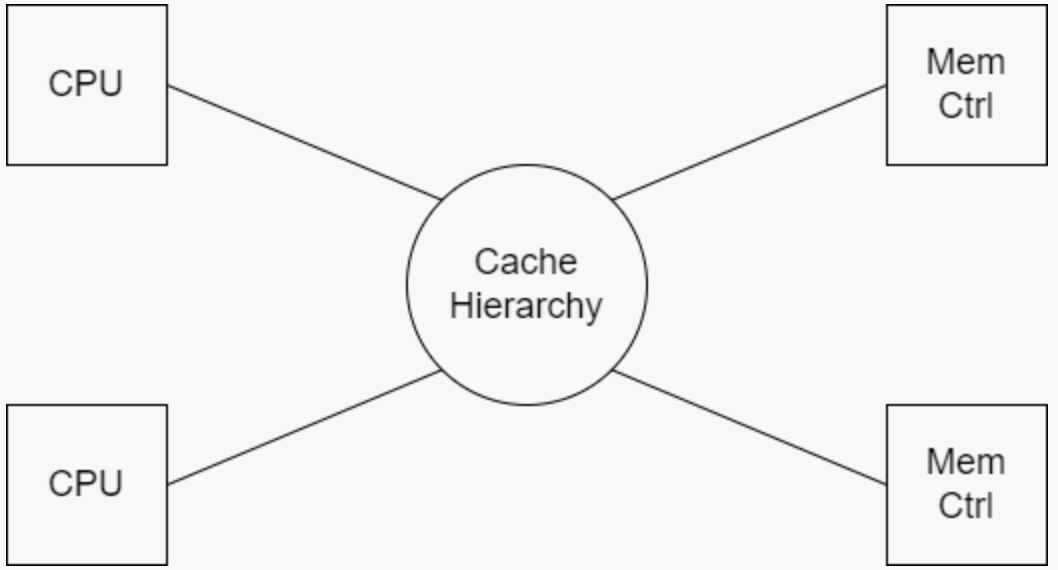
Cache Hierarchy in gem5

One of the main types of components in gem5 is the **cache hierarchy**.

In the standard library, the cache hierarchy has the `Processor` (with multiple cores) on one side and the `Memory` on the other side.

Between the cores and the caches (and the memory controllers and caches) are `Ports`.

`Ports` allow models in gem5 to send *Packets* to each other (more on this in Modeling memory objects in gem5: Ports).



Types of caches in gem5

There are two types of cache models in gem5:

1. **Classic Cache**: Simplified, faster, and less flexible
2. **Ruby**: Models cache coherence in detail

This is a historical quirk of the combination of *GEMS* which had Ruby and *m5* whose cache model we now call "classic" caches.

Ruby is a highly-detailed model with many different coherence protocols (specified in a language called "SLICC")
More on Ruby in Modeling Cache Coherence in gem5.

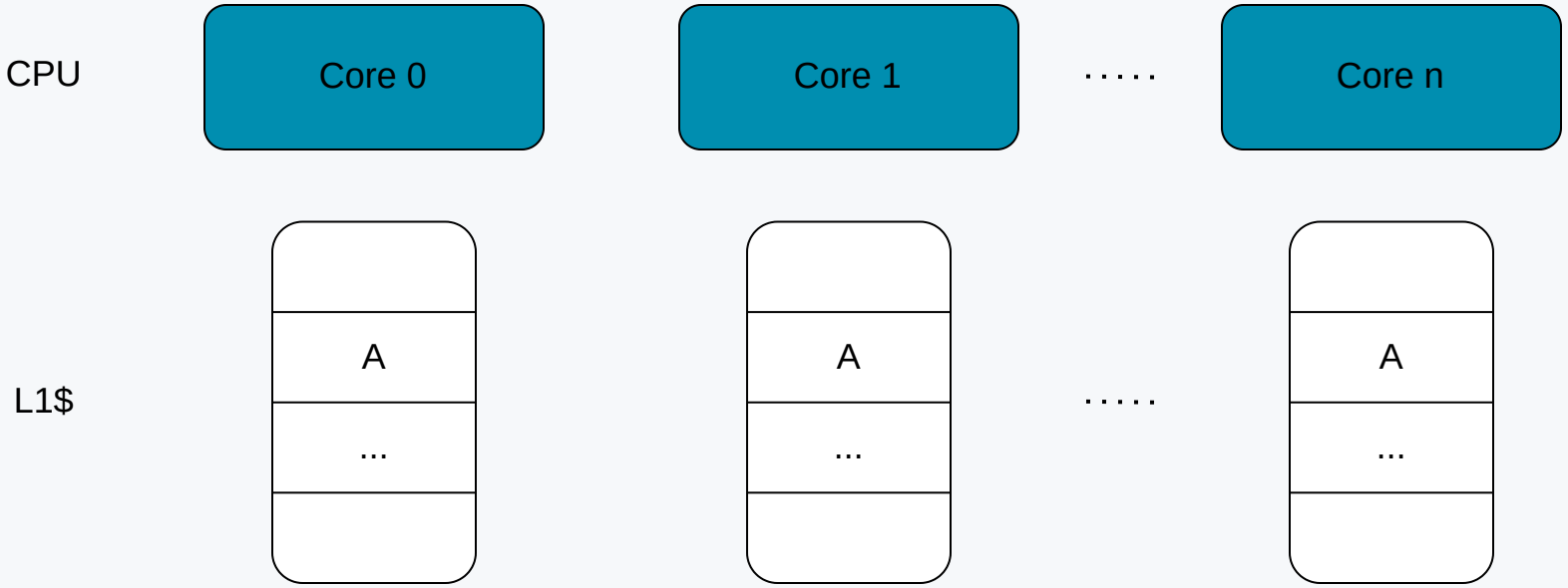
Classic caches are simpler and faster, but less flexible and detailed. The coherence protocol is not parameterized and the hierarchies and topologies are fixed.

Outline

- Background on cache coherency
- Simple Cache
  - Coherency protocol in simple cache
  - How to use simple cache
- Ruby cache
  - Ruby components
  - Example of MESI two level protocol

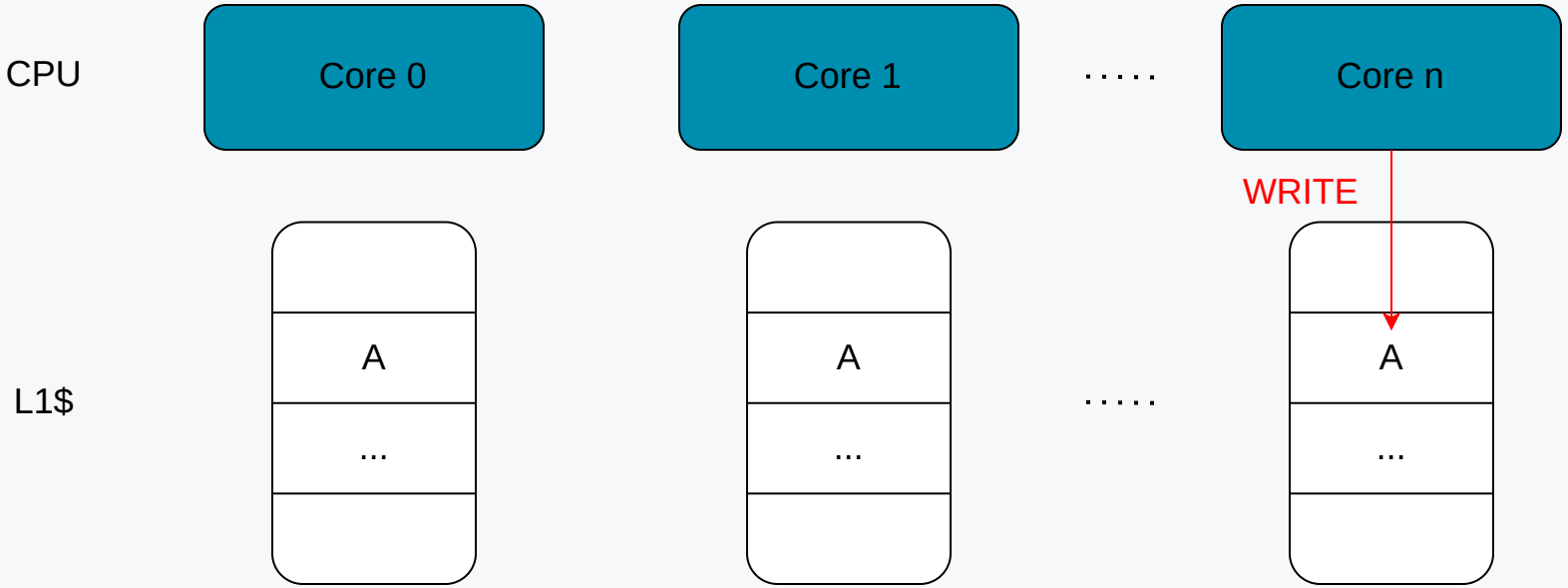
What is Coherency

A coherence problem can arise if multiple cores have access to multiple copies of a data (e.g., in multiple caches) and at least one access is a write



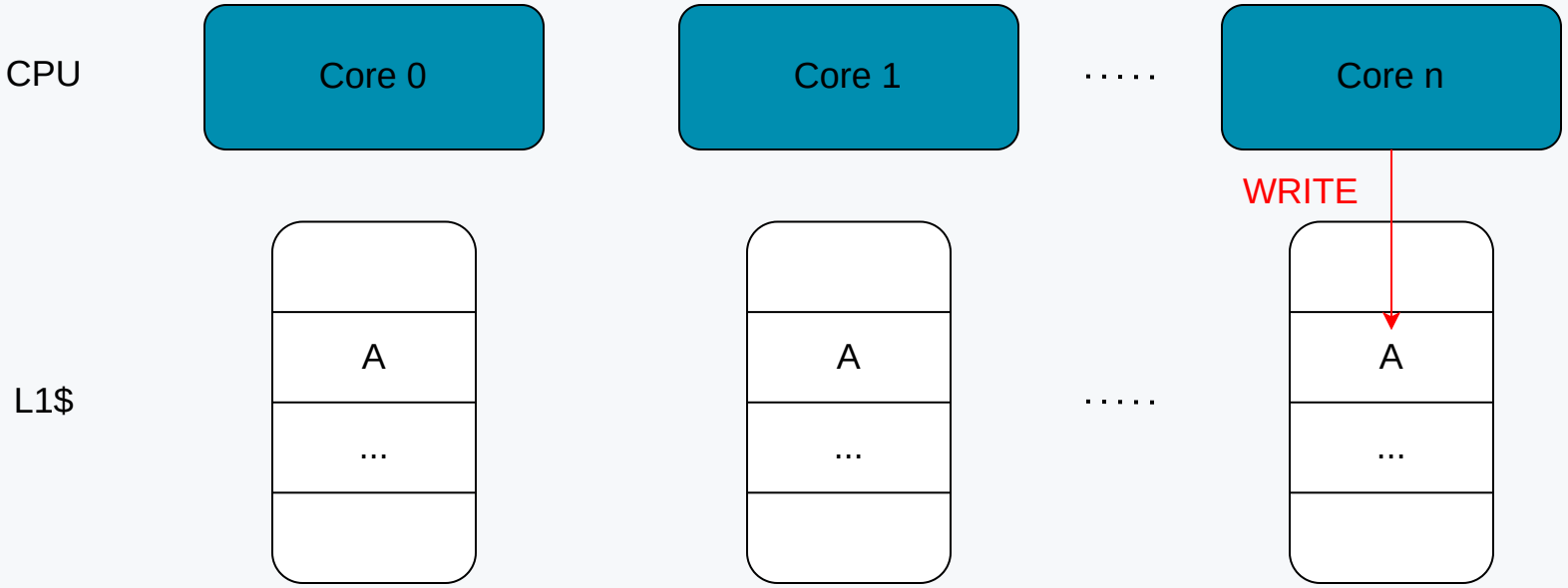
What is Coherency

A coherence problem can arise if multiple cores have access to multiple copies of a data (e.g., in multiple caches) and at least one access is a write



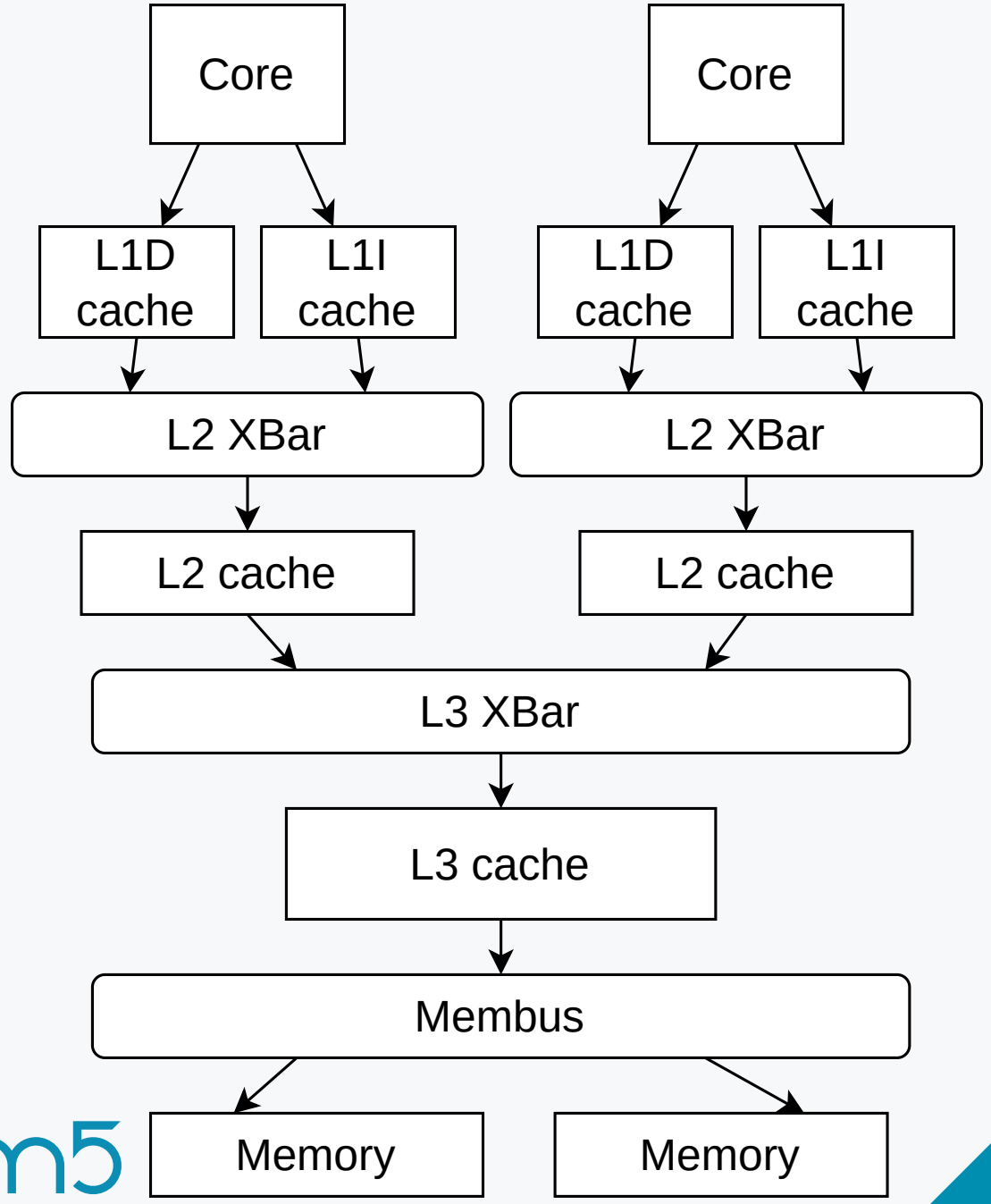
What is Coherency

A coherence problem can arise if multiple cores have access to multiple copies of a data (e.g., in multiple caches) and at least one access is a write



Classic Cache: Coherent Crossbar

Each crossbar can connect *n* cpu-side ports and *m* memory-side ports.



Exercise 1: Measure the bandwidth and latency the caches

Use the Private L1, Private L2 cache hierarchy to measure the bandwidth and latency of the cache-based system.
Use a Dual channel DDR4 memory.
Set the cache sizes to 32KiB and 256KiB for L1 and L2 respectively.

Use the traffic generator from the previous section to generate traffic.

Questions for exercise 1

A. What is the bandwidth and latency for

1. The L1 cache?
2. The L2 cache?
3. The memory?

B. What is the hit/miss ratios for the cache in your tests?

Step 1: Use a classic cache hierarchy

Use the `PrivateL1PrivateL2CacheHierarchy`.

See `PrivateL1PrivateL2CacheHierarchy` for details.

Think about what the value for `max_addr` should be for the generator to test L1, L2, and memory.

Think about whether you want to use the random or linear generator.

Step 1: Answer

```
board = TestBoard(
    generator=RandomGenerator(
        num_cores=1, max_addr=max_addr,
        rd_perc=75, duration="1ms"
    ),
    cache_hierarchy=PrivateL1PrivateL2CacheHierarchy(
        l1d_size="32KiB",
        l1i_size="32KiB",
        l2_size="256KiB",
    ),
    memory=DualChannelDDR4_2400(),
    clk_freq="3GHz",
)
```

You want to get 100% hits in the cache when testing each level and as close to 0% when you are testing the next level or memory.

Use a max address of 16384 to test the L1 cache, 131072 to test the L2 cache, and 1048576 to test the memory.

Use the random generator to maximize the hits in the cache you are testing.

Looking at "missRate::total" will show that there's 100% hits in the level we are evaluating.

Creating custom cache hierarchies

The standard library will not have all components that you will need.

Our next goal is to take a model from gem5 and wrap it in the standard library API.

Specifically, we will create a three-level cache hierarchy with private L1s, private L2s, and a shared L3 connected to multiple memory channels.

Before we dive into that exercise, we need to cover a few gem5 topics.

Classic cache models

The classic cache consists of two main models, the `Cache` and the `XBar`.

`Cache`

This is an object that represents a cache in gem5. It has a number of parameters that can be set to configure the cache.
The cache has a `cpu_side` and a `mem_side` that can be connected to other components.

`XBar`

This is a crossbar that can connect multiple ports together. It has a `cpu_side_ports` and a `mem_side_ports` that can be connected to other components.
The CPU-side and memory-side can connect to multiple objects.

gem5 Ports

Ports

Ports are the way that gem5 models communicate with each other. They are used to send packets between models.
We will see more about ports in the Modeling memory objects in gem5: Ports section.

Using ports

In gem5, to connect two ports together, you just need to use the `=` operator.

```
cache = Cache()
xbar = XBar()

cache.mem_side = xbar.cpu_side_ports
```

Specializing gem5 models

In gem5, you can specialize a model by specifying the parameters that you want to change.

For instance, if you want to create a cache with a size of 32 KiB and an associativity of 4, you can do the following:

```
class MyCache(Cache):
    def __init__(self):
        super().__init__()
        self.size = "32KiB"
        self.assoc = 4
```

You can override any of the *parameters* in the `Cache` class this way.

Exercise 2: Create a three level cache hierarchy

Create a three level cache hierarchy with private L1s, private L2s, and a shared L3 connected to multiple memory channels as shown on the previous slide.
Run your tests with all cores.

Note: For some of these steps, you'll just have to take our word for it. There are many things that have "legacy" reasons.

Questions for exercise 2:

What is the bandwidth and latency for

1. The L1 cache?
2. The L2 cache?
3. The L3 cache?
4. The memory?

Step 2: declare the hierarchy and add a membus

(Note: Step 1 is exercise 1.)

Open `materials/02-Using-gem5/04-cache-hierarchies/three_level.py`
We are creating a new class which will implement an abstract class `AbstractCacheHierarchy`.
By implementing this class, we can use it in the standard library as a cache hierarchy in our test board (or even in a CPU-based board).

The constructor is already provided.

Add a membus to the constructor

This will be your first time using a "raw" gem5 model (i.e., not something in the stdlib).

For this, use a `SystemXBar` with a width of 64.
See `SystemXBar` in /gem5/src/mem/xbar.py.

Step 2: Answer

```
self.membus = SystemXBar(width=64)
```

This will be what connects the cache to memory.

We will make it 64 Bytes wide (as wide as the cache line) so that it's maximum bandwidth.

Step 3: Implement the hierarchy interface

The board needs to be able to get the port to connect to memory.

You will be implementing some of the abstract functions from `AbstractCacheHierarchy`.

See `AbstractCacheHierarchy` for details.

You need to implement `get_mem_side_port` and `get_cpu_side_port` in this step.

Step 3: Answer

```
def get_mem_side_port(self):
    return self.membus.mem_side_ports
```

The "cpu_side_port" is used for coherent IO access from the board.

```
def get_cpu_side_port(self):
    return self.membus.cpu_side_ports
```

Step 4: Implement `incorporate_cache`

The main function is `incorporate_cache` which is called by the board after the `Processor` and `Memory` are ready to be connected together.

In the `incorporate_cache` function, we will create the caches and connect them together.

First, (given below) we will connect the system port.

Then, we will connect the memory ports to the memory bus.
You can get the list of ports for memory from the `MemorySystem` object.
Then, we can connect all of these ports to the memory side of memory bus.

Then, we will create the L3 crossbar based on the L2 crossbar parameters.

```
def incorporate_cache(self, board):
    board.connect_system_port(self.membus.cpu_side_ports)
```

Step 4: Answer

```
# Connect the memory system to the memory port on the board.
for _, port in board.get_memory().get_mem_ports():
    self.membus.mem_side_ports = port

# Create an L3 crossbar
self.l3_bus = L2XBar()
```

Here, we connect all of the memory ports to the membus.
Note that for gem5 ports, to connect things together, you just have to use the `=` operator.

We are using a "L2XBar" for the L3 crossbar because "L2XBar" is just a crossbar that has been configured to not be the system xbar.

Step 5: Create the core clusters

Let's move away from the `incorporate_cache` function for a moment and create the core clusters.

Our goal is that each core cluster should have a core connected to its private caches (L1I, L1D, L2) and the L2.
We'll then connect the L2 to the L3 via the L3 crossbar.

You'll be modifying the `_create_core_cluster` function.

Step 5: More details

A `SubSystem` is a special object that can hold multiple objects (with a shared clock/voltage domain).

1. Create a `SubSystem` object in the `_create_core_cluster` function.
2. Create the L1I and L1D caches as sub-objects of the cluster subsystem.
3. Create an L2 cache as a sub-object of the cluster subsystem.
4. Create an L2 crossbar (`L2XBar`) as a sub-object of the cluster subsystem.
5. Use the `core.connect_icache` and `core.connect_dcache` functions to connect the L1I and L1D caches to the core.
6. Connect the L1I and L1D caches to the L2 crossbar.
7. Connect the L2 cache to the L2 crossbar.
8. Connect the L2 cache to the L3 bus.

Use the `L1DCache`, `L1ICache` and `L2Cache` objects from the standard library.

> Note: there is extra code in this function. Ignore this for now.

Step 5: Answer

Since each core is going to have many private caches, let's create a cluster.
In this cluster, we will create L1I/D and L2 caches, the L2 crossbar and connect things.

```
def _create_core_cluster(self, core, l3_bus, isa):
    cluster = SubSystem()
    cluster.l1dcache = L1DCache(size=self._l1d_size, assoc=self._l1d_assoc)
    cluster.l1icache = L1ICache(
        size=self._l1i_size, assoc=self._l1i_assoc, writeback_clean=False
    )
    core.connect_icache(cluster.l1icache.cpu_side)
    core.connect_dcache(cluster.l1dcache.cpu_side)

    cluster.l2cache = L2Cache(size=self._l2_size, assoc=self._l2_assoc)
    cluster.l2_bus = L2XBar()
    cluster.l1dcache.mem_side = cluster.l2_bus.cpu_side_ports
    cluster.l1icache.mem_side = cluster.l2_bus.cpu_side_ports

    cluster.l2cache.cpu_side = cluster.l2_bus.mem_side_ports

    cluster.l2cache.mem_side = l3_bus.cpu_side_ports
```

Full-system specific things

You have been given some code to set up other caches, interrupts, etc. that are needed for full system simulation in x86 and Arm.

You can ignore this for now.

```
cluster.iptw_cache = MMUCache(size="8KiB", writeback_clean=False)
cluster.dptw_cache = MMUCache(size="8KiB", writeback_clean=False)
core.connect_walker_ports(
    cluster.iptw_cache.cpu_side, cluster.dptw_cache.cpu_side
)

# Connect the caches to the L2 bus
cluster.iptw_cache.mem_side = cluster.l2_bus.cpu_side_ports
cluster.dptw_cache.mem_side = cluster.l2_bus.cpu_side_ports

if isa == ISA.X86:
    int_req_port = self.membus.mem_side_ports
    int_resp_port = self.membus.cpu_side_ports
    core.connect_interrupt(int_req_port, int_resp_port)
else:
    core.connect_interrupt()

return cluster
```

Step 6: Create a cluster for each core

Back to `incorporate_cache`...

Now that we have the cluster, we can create the clusters.

Create a list of clusters, one for each of the cores in the processor.

Step 6: Answer

```
self.clusters = [
    self._create_core_cluster(
        core, self.l3_bus, board.get_processor().get_isa()
    )
    for core in board.get_processor().get_cores()
]
```

Step 7: Create an L3 cache

The standard library has L1 and L2 caches, but not an L3 cache.
So, we need to create an L3 cache.

Make the L3 cache a subclass of the `Cache` object.
Allow it to have a configurable size and associativity.

For the other parameters, use 20 cycles for the tag and data latency, 1 cycle for the response latency, 20 MSHRs, 12 targets per MSHR, and set `writeback_clean` to `False`.
Set the "clusivity" to "mostly_incl". In other words, this will be a mostly inclusive cache.

Step 7: Answer

```
class L3Cache(Cache):
    def __init__(self, size, assoc):
        super().__init__()
        self.size = size
        self.assoc = assoc
        self.tag_latency = 20
        self.data_latency = 20
        self.response_latency = 1
        self.mshrs = 20
        self.tgts_per_mshr = 12
        self.writeback_clean = False
        self.clusivity = "mostly_incl"
```

Step 8: Connect the L3 cache

Again, back to `incorporate_cache`...

Create an instance of your L3 cache object, pass through the size and associativity, then connect the L3 cache to the L3 crossbar and the memory bus.

Also, add this code at the end of the `incorporate_cache` function. (Trust me.)

```
if board.has_coherent_io():
    self._setup_io_cache(board)
```

Step 8: Answer

```
self.l3_cache = L3Cache(size=self._l3_size, assoc=self._l3_assoc)

# Connect the L3 cache to the system crossbar and L3 crossbar
self.l3_cache.mem_side = self.membus.cpu_side_ports
self.l3_cache.cpu_side = self.l3_bus.mem_side_ports
```

Step 9: Re run the test from Exercise 1 with 3 levels

Go back to the `test-cache.py` file that you created and import your new cache hierarchy.

Extend this to also be able to test the performance of the L3 cache.

Answers to questions

What is the bandwidth and latency for

1. The L1 cache?
2. The L2 cache?
3. The L3 cache?
4. The memory?

Classic Cache: Parameters

- src/mem/cache/Cache.py
  - src/mem/cache/cache.cc
  - src/mem/cache/noncoherent_cache.cc

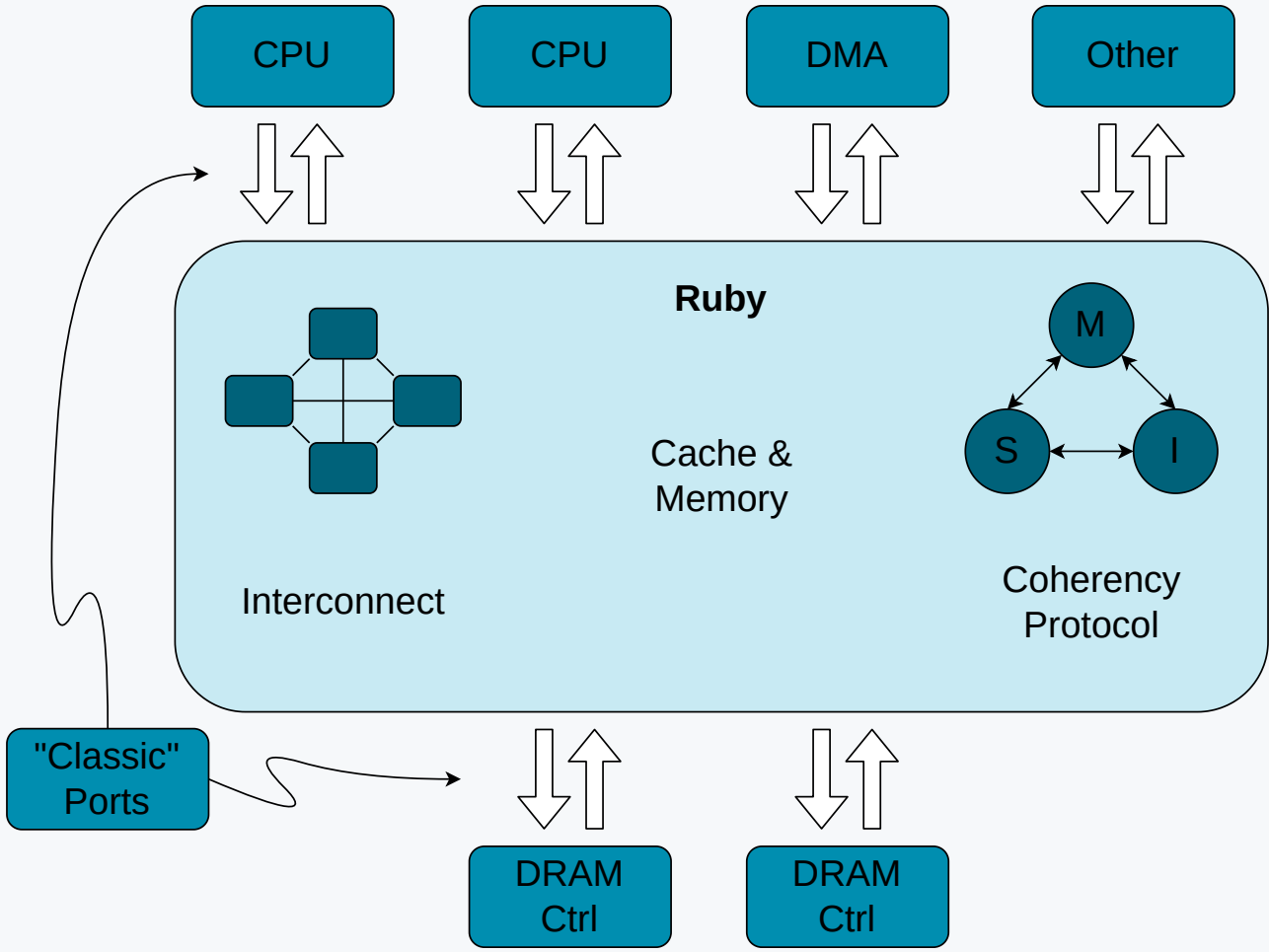
Parameters:

- size
- associativity
- number of miss status handler register (MSHR) entries
- prefetcher
- replacement policy

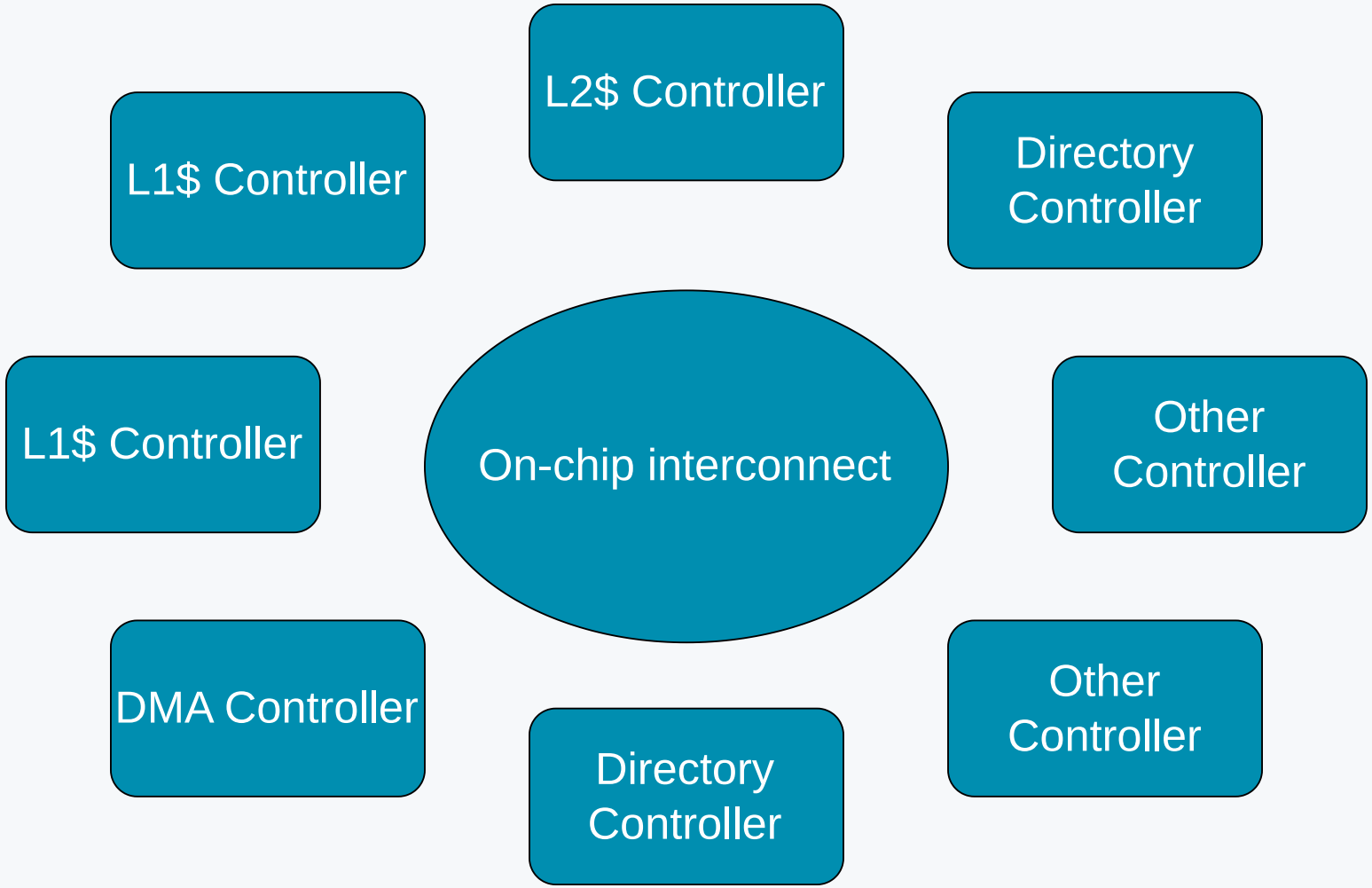
Ruby

Ruby Cache

1. Coherence Controller
2. Caches + Interface
3. Interconnect



Ruby


L2$ Controller
L1$ Controller
Directory Controller
L1$ Controller
On-chip interconnect
Other Controller
DMA Controller
Directory Controller
Other Controller

Ruby Components

- **Controller models** (cache controller, directory controller)
- **Controller topology** (Mesh, all-to-all, etc.)
- **Network models**
- **Interface** (classic ports)

Ruby Cache: Controller Models

Code for controllers is "generated" via SLICC compilers

We'll see much more detail in Modeling Cache Coherence in gem5.

Ruby Cache: Example

Let's do an example using the MESI protocol and see what new stats we can get with Ruby.

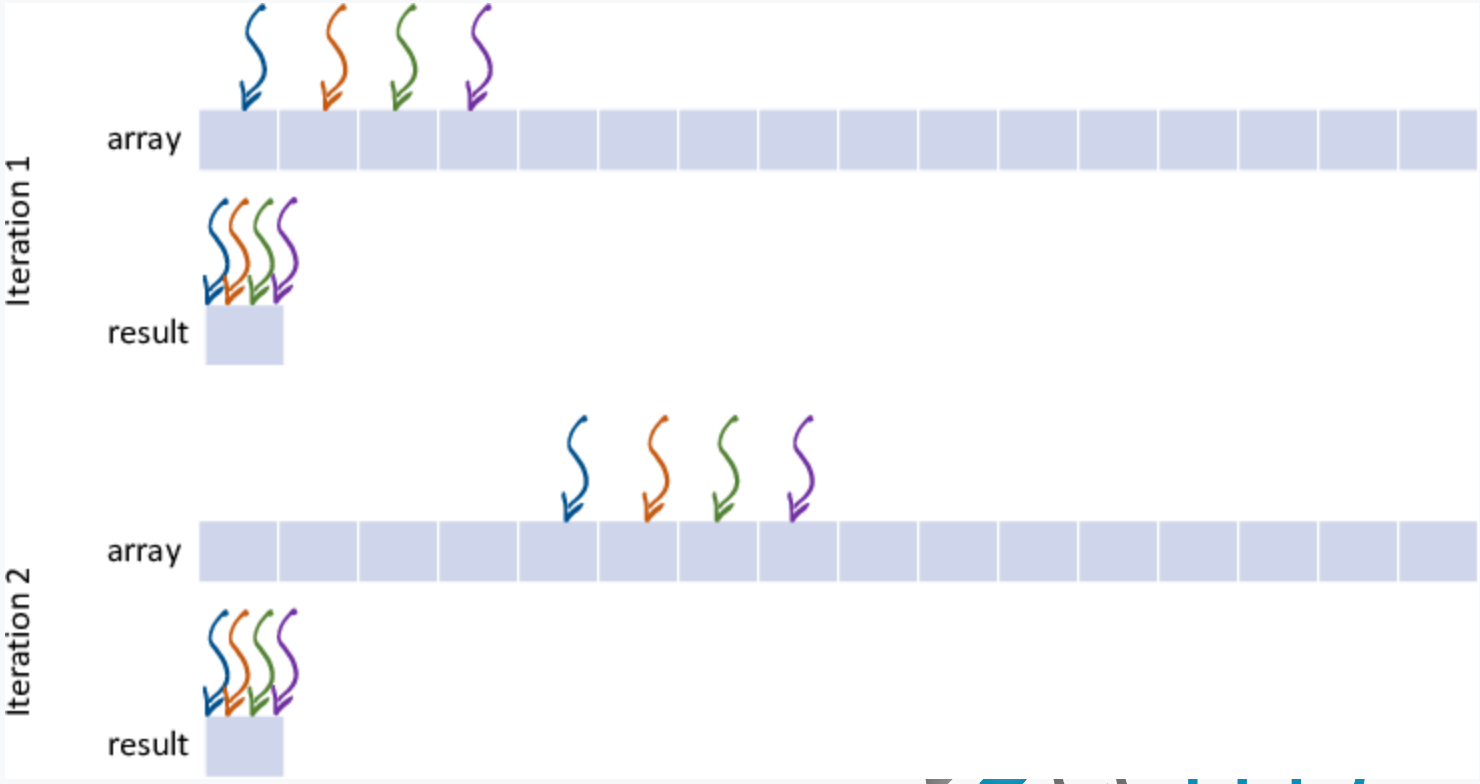
We're going to look at some different implementations of a parallel algorithm (summing an array).

```
parallel_for (int i=0; i < length; i++) {
    *result += array[i];
}
```

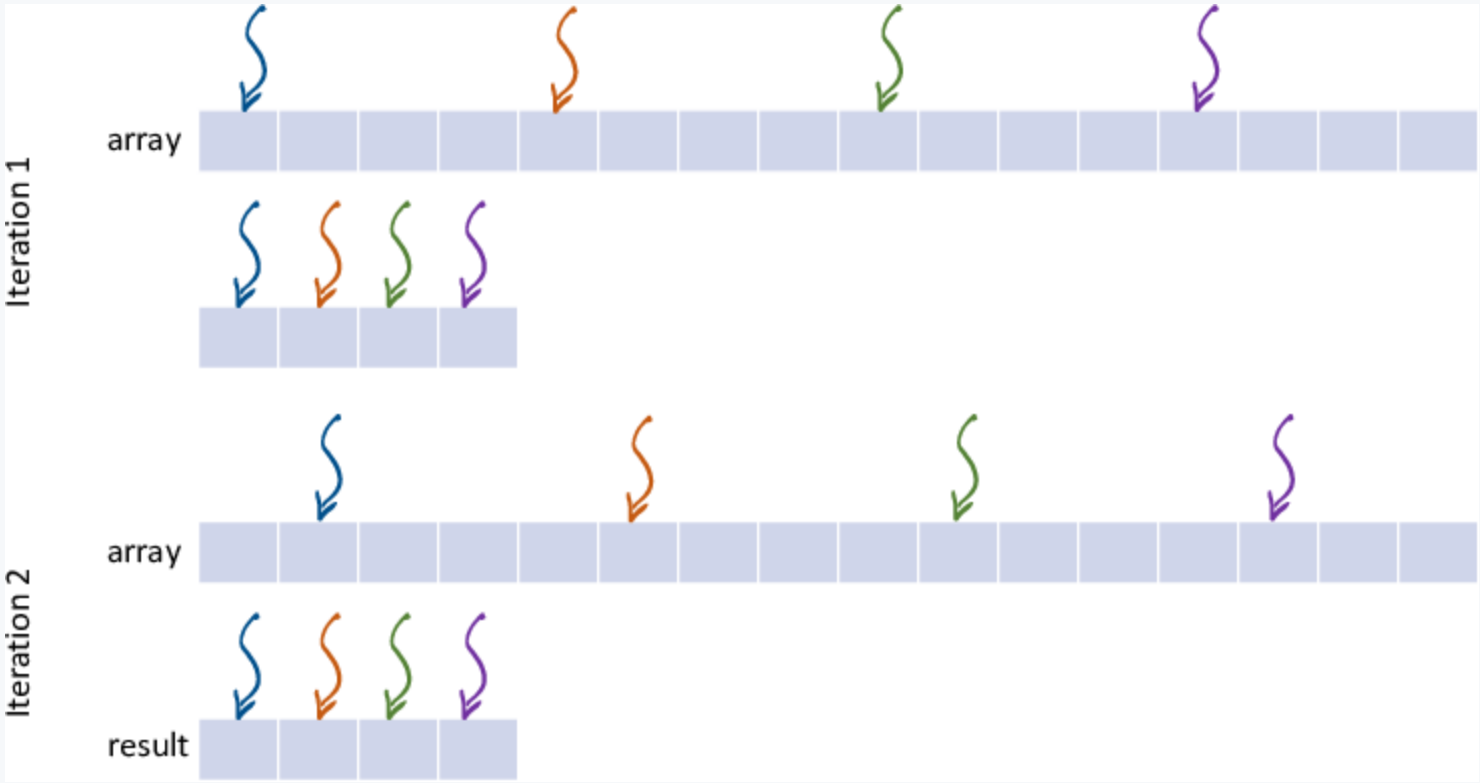
Different implementations: Naive

Three different implementations: Naive, false sharing on the output, and chunking with no false sharing.

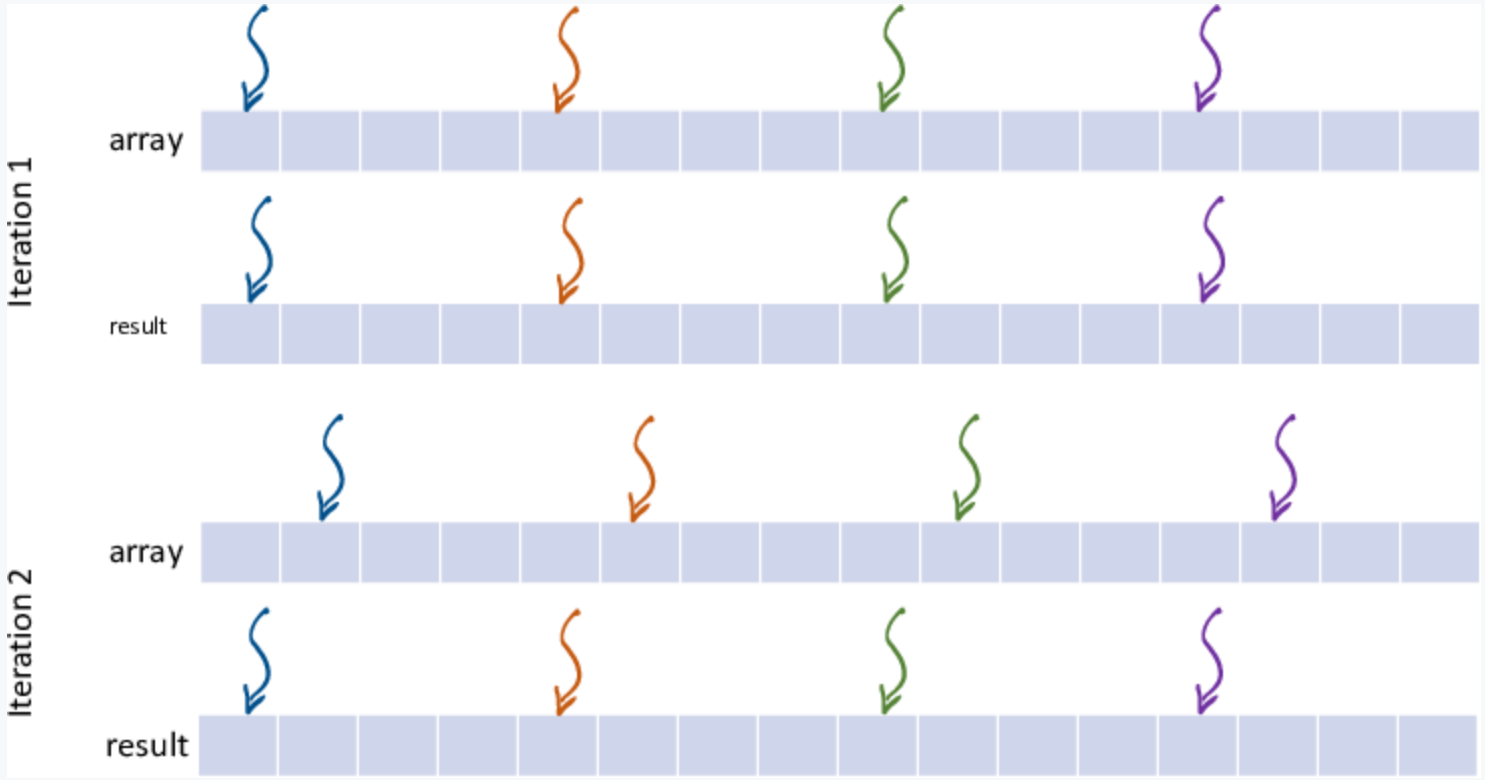
"Naive" implementation



False sharing



Chunking and no false sharing



Using Ruby

We can use Ruby to see the difference in cache behavior between these implementations.

Run the script `materials/02-Using-gem5/05-cache-hierarchies/ruby-example/run.py`.

```
gem5-mesi --outdir=m5out/naive run.py naive
```

```
gem5-mesi --outdir=m5out/false_sharing run.py false_sharing
```

```
gem5-mesi --outdir=m5out/chunking run.py chunking
```

Stats to compare

Compare the following stats:

The time it took in simulation and the read/write sharing

- `board.cache_hierarchy.ruby_system.L1Cache_Controller.Fwd_GETS`: Number of times things were read-shared
- `board.cache_hierarchy.ruby_system.L1Cache_Controller.Fwd_GETX`: Number of times things were write-shared

(Note: Ignore the first thing in the array for these stats. It's a long story...)

We'll cover more about how to configure Ruby in Modeling Cache Coherence in gem5.

Summary

- Cache hierarchies are a key part of gem5
- Classic caches are simpler and faster
- Classic caches are straightforward to configure and use
- Ruby caches are more detailed and can model cache coherence
- We can use Ruby to compare different cache behaviors