Learning gem5

Modeling Cache Coherence with Ruby and SLICC

Jason Lowe-Power

http://learning.gem5.org/

https://faculty.engineering.ucdavis.edu/lowepower/

gem5 history

M5 + GEMS

M5: “Classic” caches, CPU model, master/slave port interface

GEMS: Ruby + network

Learning gem5
http://learning.gem5.org/

Part 1: Getting started with gem5

Part 2: Modifying and extending gem5

Part 3: The Ruby cache coherence model

Part 4: Full system simulation

…. More coming

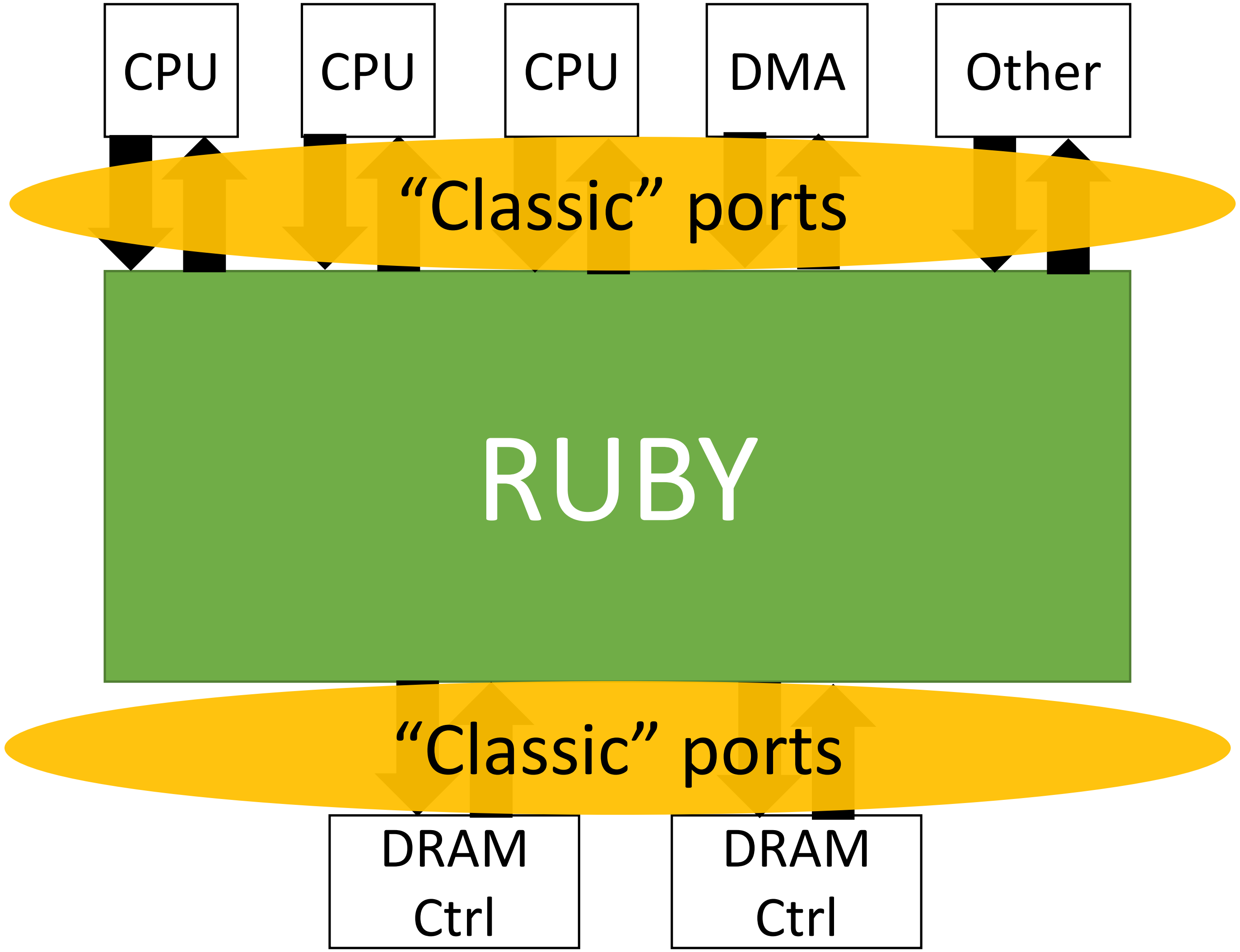
Outline

Ruby overview

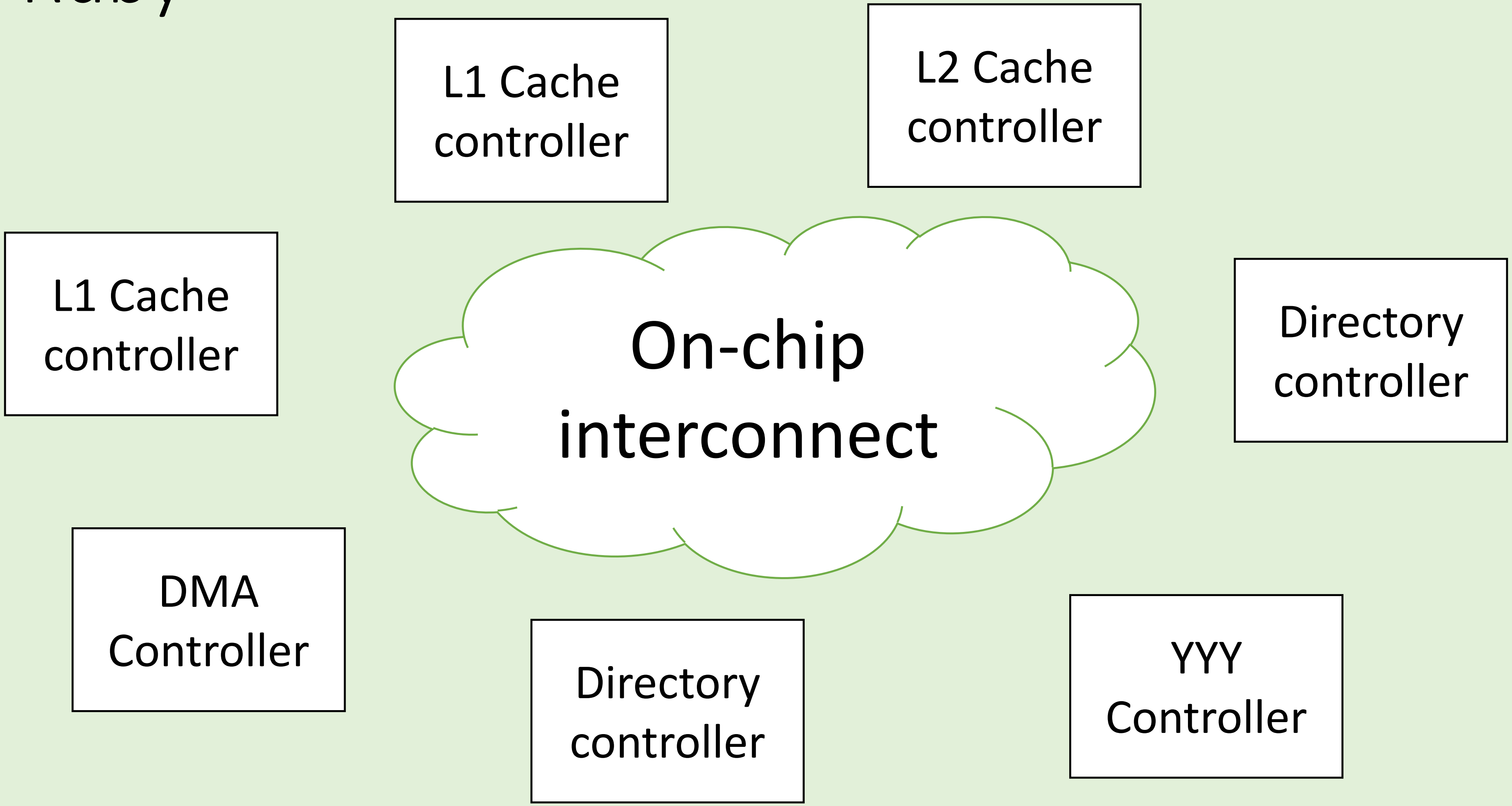
SLICC controller details

Configuring Ruby

A few other small things


CPU
CPU
CPU
DMA
Other
“Classic” ports
RUBY
“Classic” ports
DRAM Ctrl
DRAM Ctrl

Ruby


L1 Cache controller
L2 Cache controller
L1 Cache controller
On-chip interconnect
Directory controller
DMA Controller
Directory controller
YYY Controller

Ruby components

Controller models (e.g., caches)

Controller topology (how are ca

Network model (e.g., on-chip r

- Covered in more detail in the n

Main goal

Flexibility, not **usability**

Interface (“classic” ports in/out)

Controller Models

Implemented in SLICC

Code for controllers is "generated" via SLICC compiler

SLICC: Specification Language including Cache Coherence

SLICC original purpose

TABLE 8.1: MSI Directory Protocol—Cache Controller

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}										
IS^D	stall	stall	stall			stall		-/S		-/S		
IM^{AD}	stall	stall	stall	stall	stall			-/M	-/IM^A	-/M	ack--	
IM^A	stall	stall	stall	stall	stall						ack--	-/M
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^A			send Inv-Ack to Req/I						
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A	-/M	ack--	
SM^A	hit	stall	stall	stall	stall						ack--	-/M
M	hit	hit	send PutM+data to Dir/MI^A	send data to Req and Dir/S	send data to Req/I							
MI^A	stall	stall	stall	send data to Req and Dir/SI^A	send data to Req/II^A		-/I					
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I					
II^A	stall	stall	stall				-/I					

From: *A Primer on Memory Consistency and Cache Coherence*
Daniel J. Sorin, Mark D. Hill, and David A. Wood

SLICC original purpose

**Actual output

	Load	Store	Replacement	FwdGetS	FwdGetM	Inv	PutAck	DataDirNoAcks	DataDirAcks	DataOwner	InvAck	LastInvAck	
I	a aT gS pQ / IS D	a aT gM pQ / IM AD											**I**
IS D	z	z	z			z		wd dT xLh pR / S		wd dT xLh pR / S			**IS D**
IM AD	z	z	z	z	z			wd dT xSh pR / M	wd sa pR / IM A	wd dT xSh pR / M	da pR		**IM AD**
IM A	z	z	z	z	z						da pR	dT xSh pR / M	**IM A**
S	Lh pQ	aT gM pQ / SM AD	pS / SI A			iaR d pF / I							**S**
SM AD	Lh pQ	z	z	z	z	iaR pF / IM AD		wd dT xSh pR / M	wd sa pR / SM A	wd dT xSh pR / M	da pR		**SM AD**
SM A	Lh pQ	z	z	z	z						da pR	dT xSh pR / M	**SM A**
M	Lh pQ	Sh pQ	pM / MI A	cdR cdD pF / S	cdR d pF / I								**M**
MI A	z	z	z	cdR cdD pF / SI A	cdR pF / II A		d pF / I						**MI A**
SI A	z	z	z			iaR pF / II A	d pF / I						**SI A**
II A	z	z	z				d pF / I						**II A**
	Load	**Store**	**Replacement**	**FwdGetS**	**FwdGetM**	**Inv**	**PutAck**	**DataDirNoAcks**	**DataDirAcks**	**DataOwner**	**InvAck**	**LastInvAck**	

Examples

This is a ***very*** quick overview

See http://learning.gem5.org/book/part3 for more details

Based on coherence protocols in Synthesis Lecture

A Primer on Memory Consistency and Cache Coherence

Daniel J. Sorin, Mark D. Hill, and David A. Wood

MSI-cache.sm

```
machine(MachineType:L1Cache, "MSI cache")
      : Sequencer *sequencer; // Incoming request from CPU come from this
      CacheMemory *cacheMemory; // This stores the data and cache states
      bool send_evictions; // Needed to support O3 CPU and mwait
. . .
{
. . .
}
```

MSI-cache.sm

L1Cache_Controller.py

L1Cache_Controller.cc/hh

- SimObject "declaration file" ... from AbstractController
- `bool send_evictions` -> ... of the SimObject
  `send_evictions = Param.Bool("")`
- Just a SimObject

L1Cache_Entry.cc/hh

L1Cache_State.cc/hh

L1Cache_Transitions.cc/hh

L1Cache_Wakeup.cc/hh

Others…

Important!
Never modify these files!

Cache *state machine* outline

Parameters:

- **Cache memory**: Where the data is stored
- **Message buffers**: Sending/receiving messages from network

State declarations: The stable and transient states

Event declarations: State machine events that will be "triggered"

Other structures and functions: Entries, TBEs, get/setState, etc.

In ports: Trigger *events* based on incoming messages

Actions: Execute *single* operations on cache structures

Transitions: Move from *state* to *state* and execute *actions*

Cache memory

See src/mem/ruby/structures/CacheMemory

Stores the cache data (Entry) and the state (State)

cacheProbe() returns the replacement address if cache is full

Important!
Must call setMRU on each access!

Message buffers

Declaring is confusing!

```
MessageBuffer * requestToDir, network="To", virtual_network="0", vnet_type="request";

MessageBuffer * forwardFromDir, network="From", virtual_network="1", vnet_type="forward";
```

peek(): Get the head message

pop(): Remove head message (don't forget this!)

isReady(): Is there a message?

recycle(): Move the head to the tail (better perf., but unrealisitic)

stallAndWait(): Move (stalled) message to different buffer

State declarations

```
state_declaration(State, desc="Cache
 I,       AccessPermission:Invalid, d

 // States moving out of I
 IS_D,    AccessPermission:Invalid, desc="Invalid, moving to S, waiting for data“;
 IM_AD,   AccessPermission:Invalid, desc="Invalid, moving to M, waiting for acks and data";
 IM_A,    AccessPermi

 S,       AccessPermi                                                          ne block";

 . . .
}
```

AccessPermission: Used for functional accesses

IS_D -> Read: “Invalid transitioning to Shared waiting for Data”

Event declarations

```
enumeration(Event, desc="Cache events") {
 // From the processor/sequencer/mandatory queue
 Load,           desc="Load from processor";
 Store,          desc="Store from processor";

 // Internal event (only triggered from processor requests)
 Replacement,    desc="Triggered when block is chosen as victim";

 // Forwarded request from other cache via dir on the forward network
 FwdGetS,        desc="Directory sent us a request to satisfy GetS. ";
                      "We must have the block in M to respond to this.";
 FwdGetM,        desc="Directory sent us a request to satisfy GetM. ";
 . . .
```

Other structures and functions

Entry: Declare the data structure for each entry

Block data, block state, sometimes others (e.g., tokens)

TBE/TBETable: Transient Buffer Entry

Like an MSHR, but not exactly (allocated more often)

Holds data for blocks in *transient* states

get/set State, AccessPermissions, functional read/write

Required to implement AbstractController

Usually just copy-paste from examples

In ports

Weird syntax!
Automatically populates "in_msg"
in the following block

```
in_port(forwar
 if (forward_in.isReady(clockEdge())) {
   peek(forward_in, RequestMsg) {
     Entry cache_entry := getCacheEntry(in_msg.addr);
     TBE tbe := TBEs[in_msg.addr];
     if (in_msg.Type == CoherenceRequestType:GetS) {
        trigger(Event:FwdGetS, in_msg.addr, cache_entry, tbe);
     } else
 . . .
```

Trigger() looks for a *transition*. It also ensures resources available.

Actic

Like “peek”, but populates out_msg

```
action(sendGetM, "gM", desc="Send GetM to the directory") {
 enqueue(request_out, RequestMsg, 1) {
    out_msg.addr := address;
    out_msg.Type := CoherenceRequestType:GetM;
    out_msg.D

    out_msg.M
    out_msg.R
 }
}
```

Some variables are implicit in actions. These are passed in via trigger() in in_port.
address, cache_entry, tbe

Transitions

Begin state

End state

On event

Either event

Either state

```
transition(I, Store, IM_AD) {
  allocateCacheBlock;
  allocateTBE;
  sendGetM;
  popMandatoryQueue;
}

transition({IM_AD, SM_AD}, {DataDirNoAcks, DataOwner}, M) {
  writeDataToCache;
  deallocateTBE;
  externalStoreHit;
  popResponseQueue;
}
```

More details at

http://learning.gem5.org/book/part3

Ruby config scripts

Don’t follow gem5 style closely :(

Require lots of boilerplate

Ruby config scripts

1. Instantiate the controllers

    Here is where you pass all of the options from the *.sm file

2. Create a *Sequencer* for each CPU

    More details in a moment

3. Create and connect all of the network routers

Creating the topology

Usually hidden in “create_topology” (see configs/topologies)

- **Problem**: These make assumptions about controllers
- Inappropriate for non-default protocols

Point-to-point example

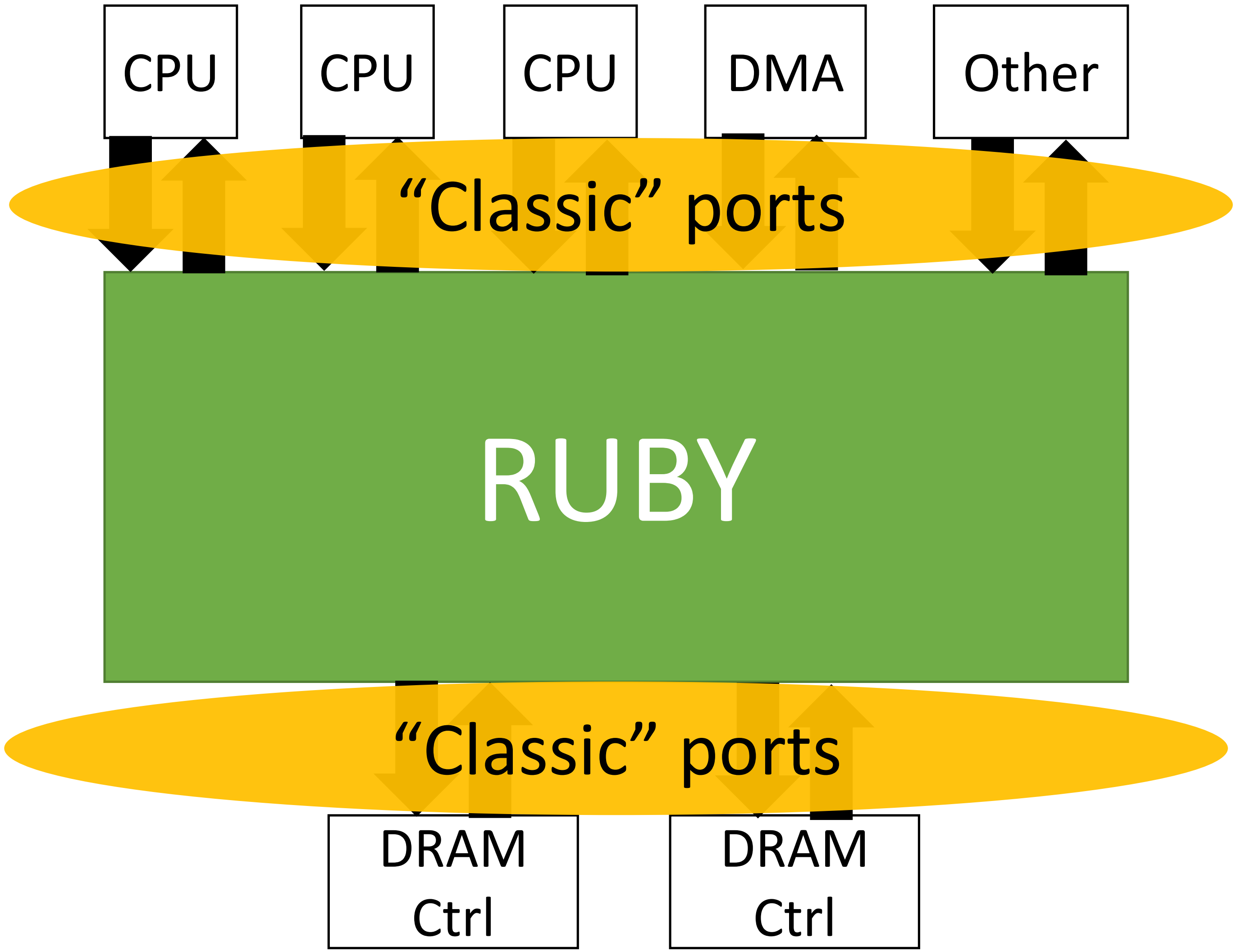
An "external" link between the controller and the network

One router per controller

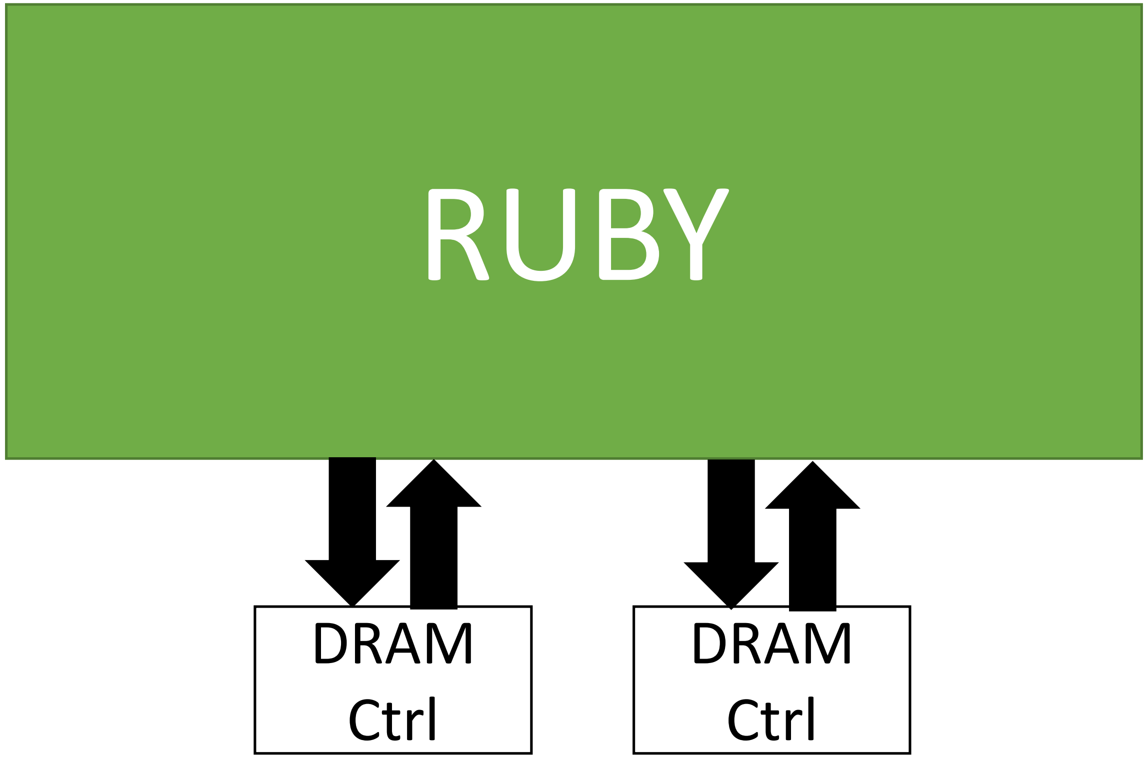
```
self.routers = [Switch(rout

self.ext_links = [                xt_node=c,
                                  f.routers[i])
                                  rollers)]


link_count = 0
self.int_links = []
for ri in self.routers:
    for rj in self.routers:
        if ri == rj: continue #
        link_count += 1
        self.int_links.append(SimpleIntLink(link_id = link_count,
                                            src_node = ri,
                                             dst_node = rj))
```

An "internal" link between each of the routers to every other router

Ports -> Ruby interface



Ruby -> Memory



Any controller can connect its “memory” port.
Usually, only “directory controllers.

You can send messages on this port in SLICC with queueMemoryRead/Write

Responses come on special message buffer (responseFromMemory)

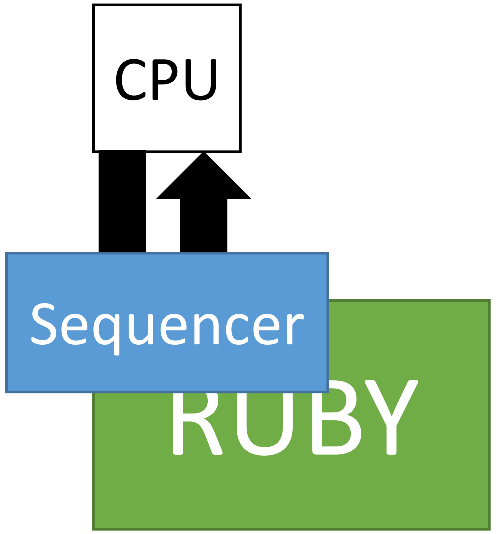
CPU->Ruby: Sequencers

Confusing: Two names, same thing: RubyPort and Sequencer

Sequencer is a MemObject (classic ports)

Converts gem5 packets to RubyRequests

New messages delivered to the "MandatoryQueue"



Where is . . . ?

Configuration

configs/network	Configuration of network models
configs/topologies	Default cache topologies
configs/ruby	Protocol config and Ruby config

Ruby config: configs/ruby/Ruby.py

Entry point for Ruby configs and helper functions

Selects the right protocol config "automatically"

Jason Lowe-Power <jason@lowepower.com>

Where is . . . ?

Don't be afraid to dig into the compiler! It's often *necessary*.

SLICC

src/mem/slicc — Code for the compiler

src/mem/ruby/slicc_interface

Structures used only in generated code

AbstractController

Where is . . . ?

src/mem/ruby/structures

Structures used in Ruby (e.g., cache memory, replace policy)

src/mem/ruby/system

Ruby wrapper code and entry point

RubyPort/Sequencer

RubySystem: Centralized information, checkpointing, etc.

Where is . . . ?

src/mem/ruby/common	General data structures, etc.
src/mem/ruby/filters	Bloom filters, etc.
src/mem/ruby/network	Network model (see next talk!)
src/mem/ruby/profiler	Profiling for coherence protocols

Current protocols (src/mem/protocol)

GPU rfo (Read for ownership GPU-CPU protocol)
GPU VIPER ("Realistic" GPU-CPU protocol)
GPU VIPER Region (HSA paper)
Garnet standalone (No coherence, just traffic injection)
MESI Three level (like two level, but with L0 cache)
MESI Two level (private L1s shared L2)
MI example (Example: Do not use for performance)
MOESI AMD (??)
MOESI CMP directory
MOESI CMP token
MOESI hammer (Like AMD hammer protocol for opteron/hyper transport)

Things not covered

Writing a coherence protocol

- Virtual networks
- Stalling requests
- Extra transient states

Debugging a coherence protocol

- RubyRandomTester + ProtocolTrace
- Other Ruby debug flags also useful

Many more details in the book:

learning.gem5.org/book

github.com/powerjg/learning_gem5

If you find a mistake:

Create an issue on github

Fork on github and fix it!

More resources

http://learning.gem5.org/book

http://gem5.org/SLICC

http://gem5.org/Ruby