The gem5 Simulator

ISCA 2011

Brad Beckmann[1] Nathan Binkert[2] Ali Saidi[3] Joel Hestness[4]
Gabe Black[5] Korey Sewell[6] Derek Hower[7]

[1] AMD Research [2] HP Labs [3] ARM, Inc. [4] University of Texas, Austin
[5] Google, Inc. [6] University of Michigan, Ann Arbor
[7] University of Wisconsin, Madison

June 5th, 2011

Outline

Introduction to gem5

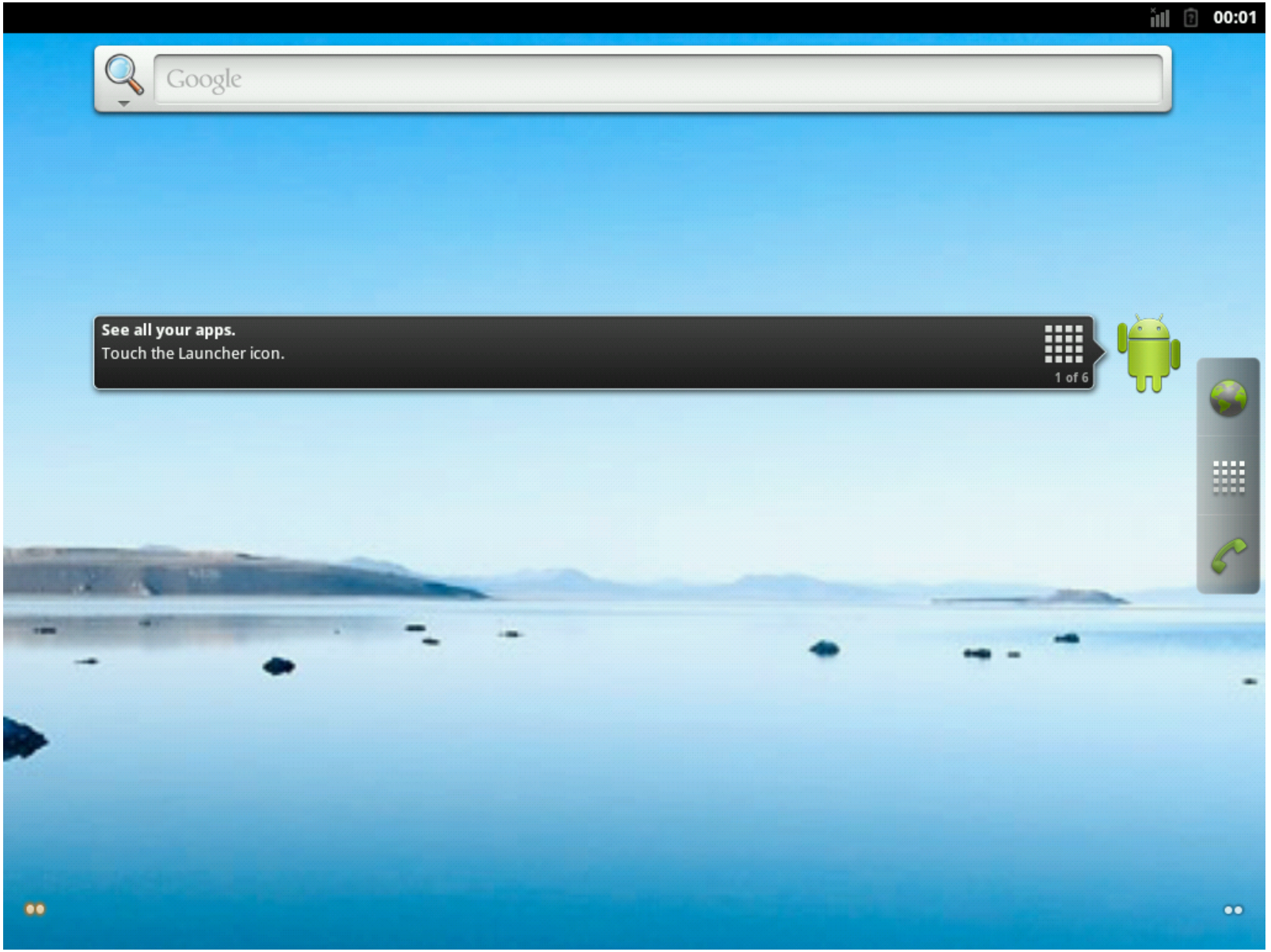
Brad Beckmann

AMD Research

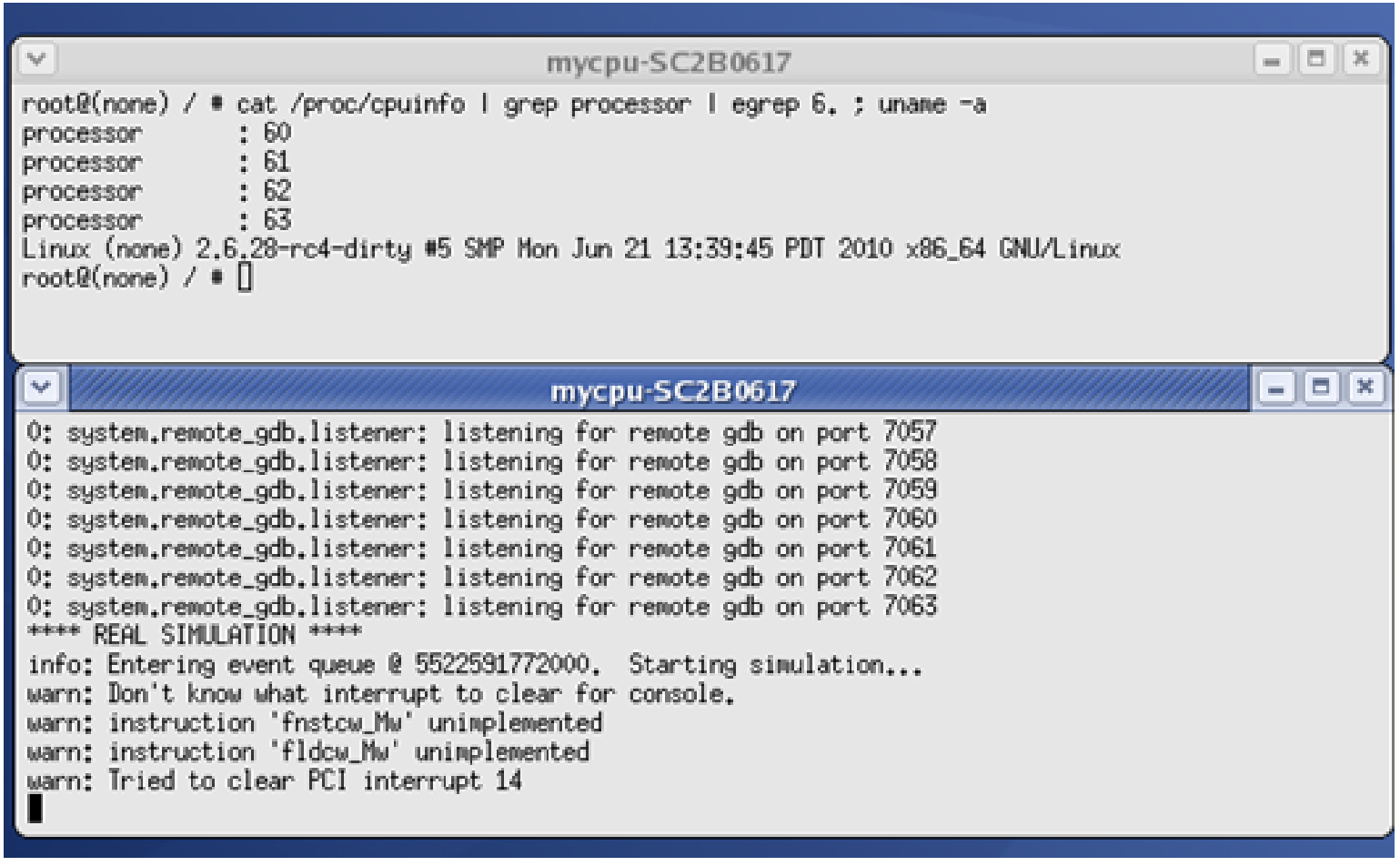
What is gem5?

- The combination of M5 and GEMS into a new simulator
  - Google scholar statistics
    - M5 (IEEE Micro, CAECW): **440** citations
    - GEMS (CAN): **588** citations
  - Best aspects of both glued together
    - M5: CPU models, ISAs, I/O devices, infrastructure
    - GEMS (essentially Ruby): cache coherence protocols, interconnect models

Android on ARM FS



64 Processor Linux on x86 FS



Main Goals

Overall Goal: *Open source community tool focused on architectural modeling*

- Flexibility
  - Multiple CPU models across the speed vs. accuracy spectrum
  - Two execution modes: System-call Emulation & Full-system
  - Two memory system models: Classic & Ruby
  - Once you learn it, you can apply to a wide-range of investigations
- Availability
  - For both academic and corporate researchers
  - No dependence on proprietary code
  - BSD license
- Collaboration
  - Combined effort of many with different specialties
  - Active community leveraging collaborative technologies

Key Features

- Pervasive object-oriented design
  - Provides modularity, flexibility
  - Significantly leverages inheritance *e.g. SimObject*
- Python integration
  - Powerful front-end interface
  - Provides initialization, configuration, & simulation control
- Domain-Specific Languages
  - ISA DSL: defines ISA semantics
  - Cache Coherence DSL (a.k.a.**SLICC**): defines coherence logic
- Standard interfaces: `Ports` and `MessageBuffers`

Capabilities

- **Execution modes**: System-call Emulation (SE) & Full-System (FS)
- **ISAs**: Alpha, ARM, MIPS, Power, SPARC, x86
- **CPU models**: AtomicSimple, TimingSimple, InOrder, and O3
- **Cache coherence protocols**: broadcast-based, directories, etc.
- **Interconnection networks**: Simple & Garnet (Princeton, MIT)
- **Devices**: NICs, IDE controller, etc.
- **Multiple systems**: communicate over TCP/IP

Cross-Product Matrix

Processor		Memory System		
CPU Model	System Mode	Classic	Ruby	
			Simple	Garnet
Atomic Simple	SE FS			
Timing Simple	SE FS			
InOrder	SE FS			
O3	SE FS			

Outline

Basics

Nate Binkert

HP Labs

Basics

- Compiling gem5
- Running gem5
- ~~*Very* brief overview of a few key concepts:~~
  - ~~Objects~~
  - ~~Events~~
  - ~~Modes~~
  - ~~Ports~~
  - ~~Stats~~

Building Executables

- Platforms
  - Linux, BSD, MacOS, Solaris, etc.
  - Little endian machines
    - Some architectures support big endian
  - 64-bit machines help a lot
- Tools
  - GCC/G++ 3.4.6+
    - Most frequently tested with 4.2-4.5
  - Python 2.4+
  - SCons 0.98.1+
    - We generally test versions 0.98.5 and 1.2.0
    - http://www.scons.org
  - SWIG 1.3.31+
    - http://www.swig.org

Compile Targets

- build/<config>/<binary>
  - configs
    - By convention, usually <isa>_<mode>
    - ALPHA_SE (Alpha syscall emulation)
    - ALPHA_FS (Alpha full system)
    - Other ISAs: ARM, MIPS, POWER, SPARC, X86
    - Sometimes followed by Ruby protocol:
    - ALPHA_SE_MOESI_hammer
    - You can define your own configs
  - binary
    - gem5.debug – debug build, symbols, tracing, assert
    - gem5.opt – optimized build, symbols, tracing, assert
    - gem5.fast – optimized build, no debugging, no symbols, no tracing, no assertions
    - gem5.prof – gem5.fast + profiling support

Modes

- gem5 has two fundamental modes
- Full system (FS)
  - For booting operating systems
  - Models bare hardware, including devices
  - Interrupts, exceptions, privileged instructions, fault handlers
- Syscall emulation (SE)
  - For running individual applications, or set of applications on MP/SMT
  - Models user-visible ISA plus common system calls
  - System calls emulated, typ. by calling host OS
  - Simplified address translation model, no scheduling
- Selected via compile-time option
  - Vast majority of code is unchanged, though

Outline

Outline

Outline

Outline

Multiple Architecture Support

Gabe Black

Google, Inc.

Overview

- Tour of the ISAs
- Parts of an ISA
- Decoding and instructions

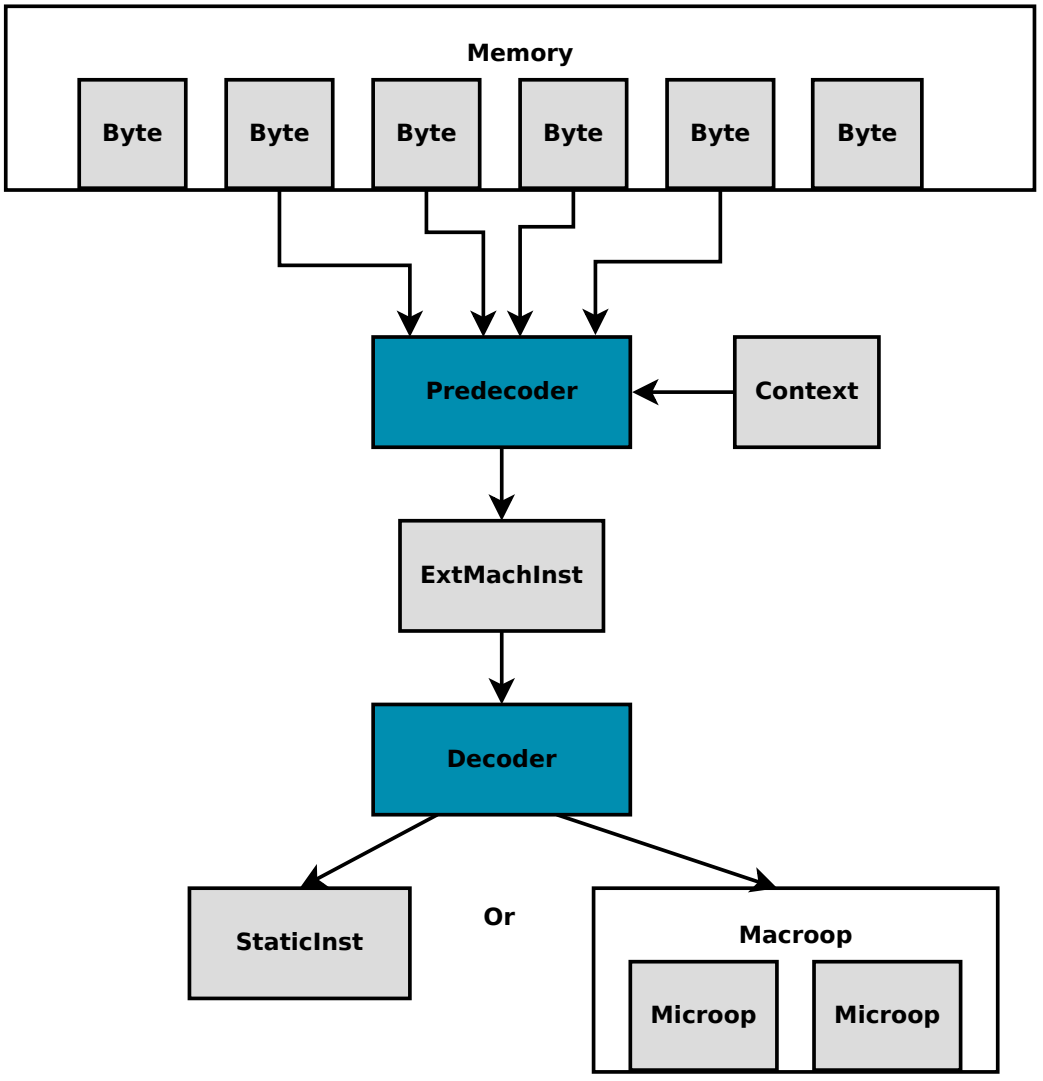
ISA Support

- Full-System & Syscall Emulation
  - Alpha
  - ARM
  - SPARC
  - x86
- Syscall Emulation
  - MIPS
  - POWER

Parts of an ISA

- Parameterization
  - Number of registers
  - Endianness
  - Page size
- Specialized objects
  - TLBs
  - Faults
  - Control state
  - Interrupt controller
- Instructions
  - Instructions themselves
  - Decoding mechanism

Instruction decode process



ISA Description Language

`src/arch/isa_parser.py, src/arch/*/isa/*`

- Custom domain-specific language
- Defines decoding & behavior of ISA
- Generates C++ code
  - Scads of **StaticInst** subclasses
  - **decodeInst ()** function
    - Maps machine instruction to **StaticInst** instance
  - Multiple scads of **execute()** methods
    - Cross-product of CPU models and **StaticInst** subclasses

Key Features

- Very compact representation
  - Most instructions take 1 line of C code
  - Alpha: 3437 lines of isa description $\rightarrow$ 39K lines of C++
    - $\sim$15K generic decode, $\sim$12K for each of 2 CPU models
  - Characteristics auto-extracted from C
    - source, dest regs; func unit class; etc.
  - `execute()` code customized for CPU models
- Thoroughly documented (for us, anyway)
  - See wiki pages

Outline

CPU Modeling

Korey Sewell

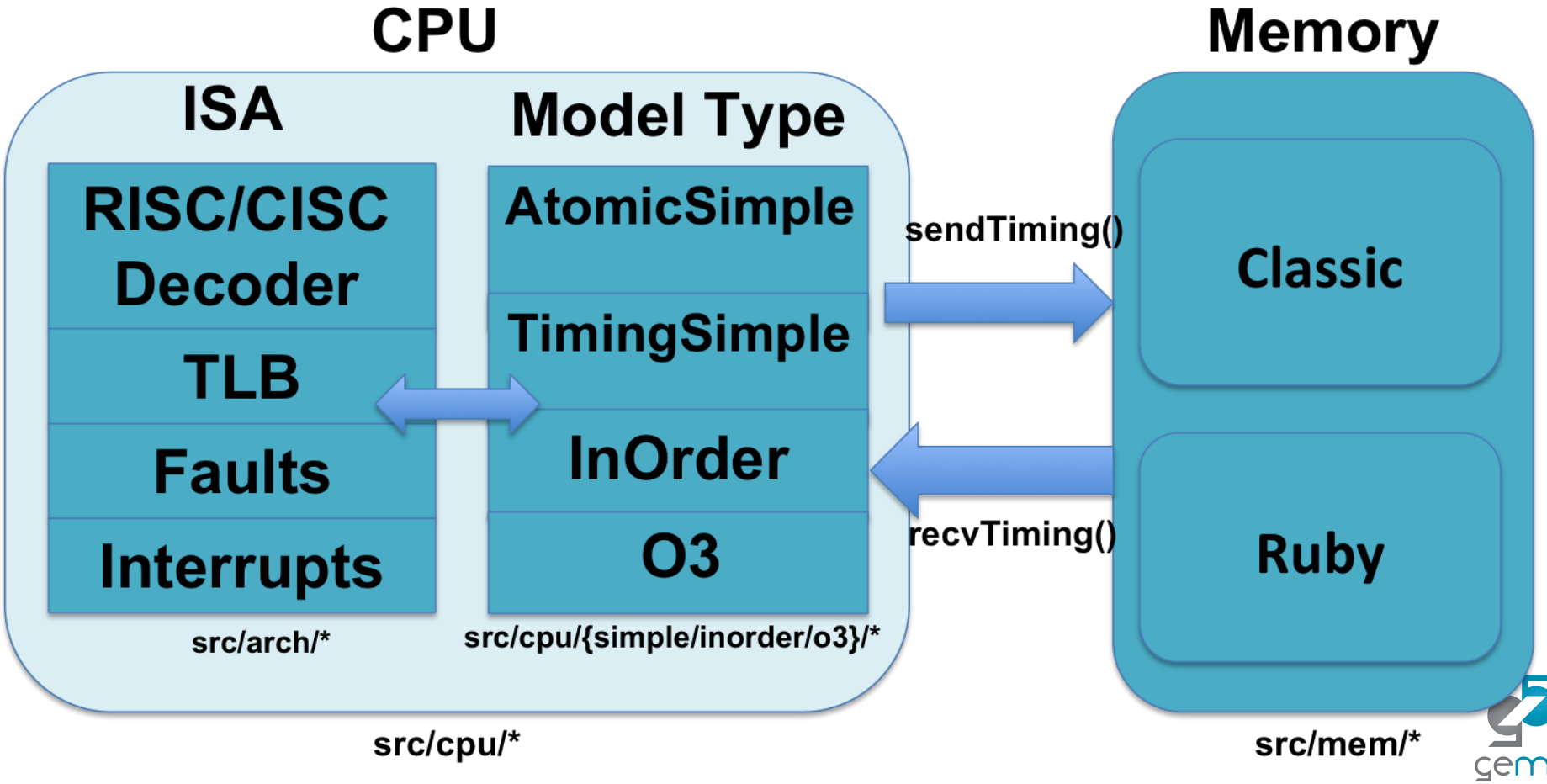
University of Michigan, Ann Arbor

Overview

- High Level View
- Supported CPU Models
  - AtomicSimpleCPU
  - TimingSimpleCPU
  - InOrderCPU
  - O3CPU
- ~~CPU Model Internals~~
  - ~~Parameters~~
  - ~~Time Buffers~~
  - ~~Key Interfaces~~

CPU Models - System Level View

CPU Models are designed to be “hot pluggable” with arbitrary ISAs and Memory Systems



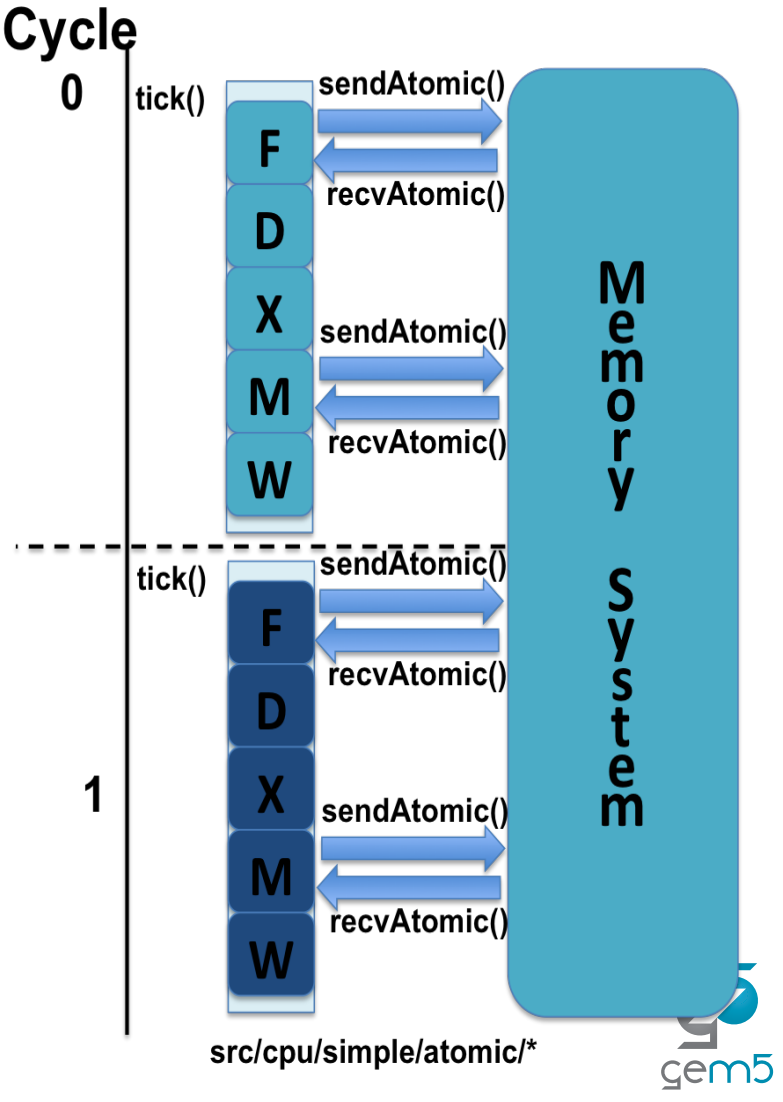
Supported CPU Models

`src/cpu/*.hh,cc`

- Simple CPUs
  - Models Single-Thread 1 CPI Machine
  - Two Types: AtomicSimpleCPU and TimingSimpleCPU
  - Common Uses:
    - Fast, Functional Simulation: 2.9 million and 1.2 million instructions per second on the “twolf” benchmark
    - Warming Up Caches
    - Studies that do not require detailed CPU modeling
- Detailed CPUs
  - Parameterizable Pipeline Models w/SMT support
  - Two Types: InOrderCPU and O3CPU
  - “Execute in Execute”, detailed modeling
  - Slower than SimpleCPUs: 200K instructions per second on the “twolf” benchmark
    - Models the timing for each pipeline stage
    - Forces both timing and execution of simulation to be accurate
    - Important for Coherence, I/O, Multiprocessor Studies, etc.

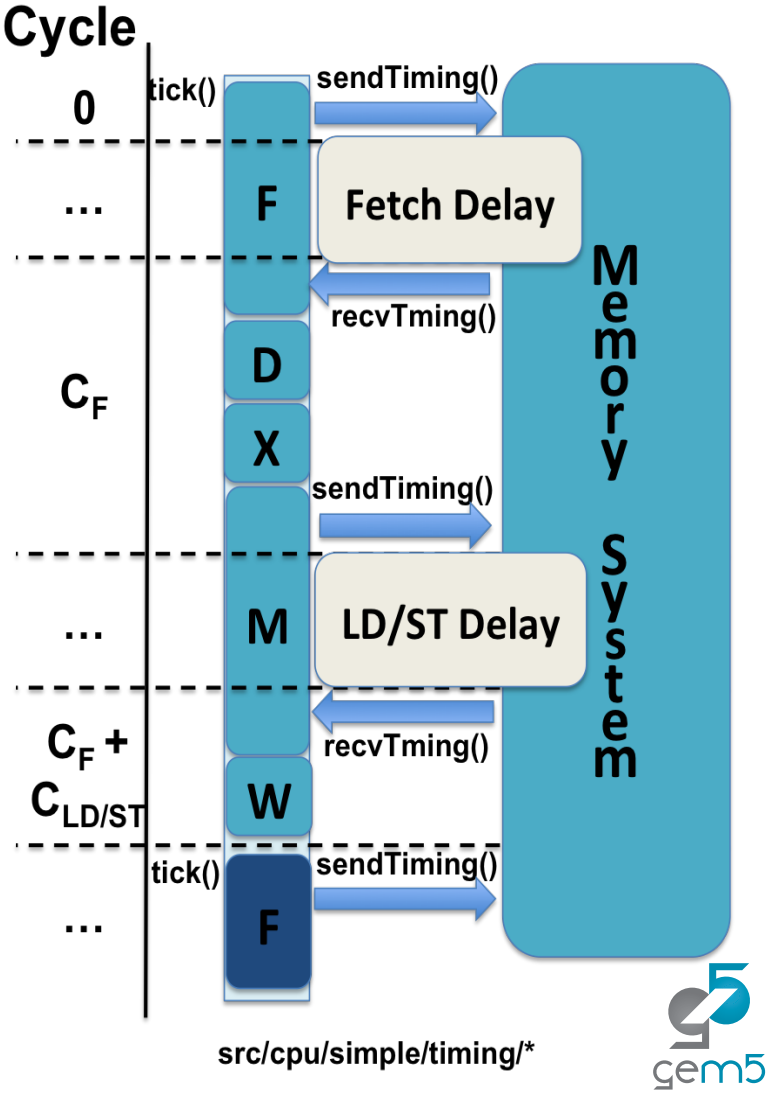
AtomicSimpleCPU `src/cpu/simple/atomic/*.hh,cc`

- On every CPU “tick()”, perform all necessary operations for an instruction
- Memory accesses are atomic
- Fastest functional simulation



TimingSimpleCPU `src/cpu/simple/timing/*.hh,cc`

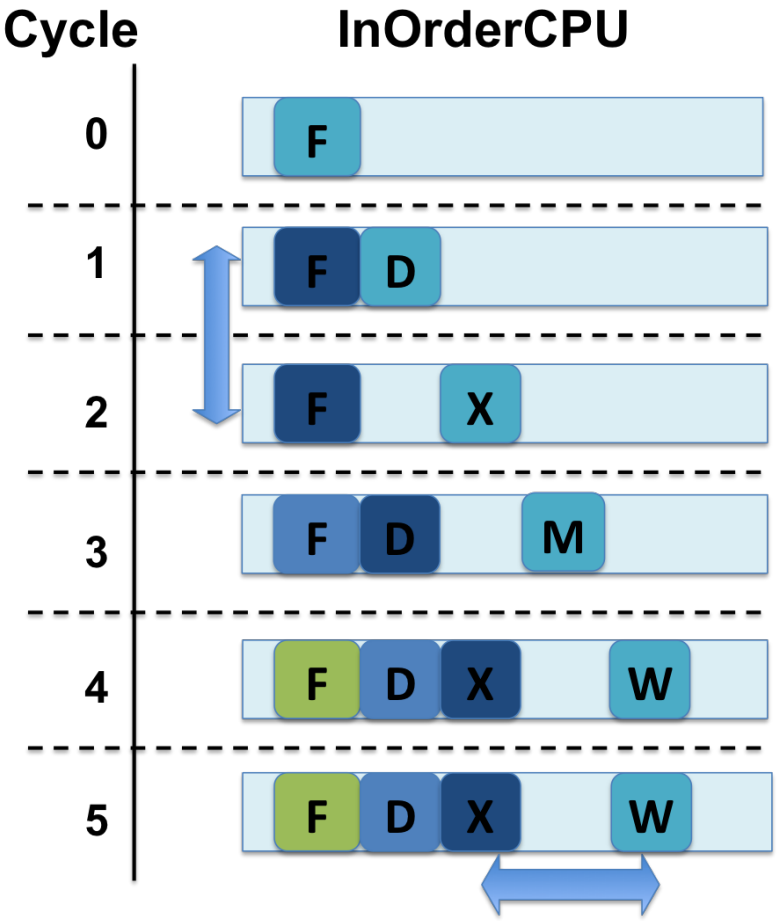
- Memory accesses use timing path
- CPU waits until memory access returns
- Fast, provides some level of timing



InOrder CPU Model

`src/cpu/inorder/*.hh,cc`

- Detailed in-order CPU
- *InOrder* is a new feature to the gem5 Simulator
  - Default 5-stage pipeline
    - Fetch, Decode, Execute, Memory, Writeback



InOrder CPU Model `src/cpu/inorder/*.hh,cc`

- Detailed in-order CPU
  - Default 5-stage pipeline
    - Fetch, Decode, Execute, Memory, Writeback
  - Key Resources
    - CacheUnit, ExecutionUnit, BranchPredictor, etc.
  - Key Parameters
    - Pipeline Stages, Hardware Threads
- Implementation: Customizable Set of Pipeline Components
  - Pipeline stages interact with *Resource Pool*
  - Pipeline defined through *Instruction Schedules*
    - Each instruction type defines what resources they need in a particular stage
    - If an instruction can't complete all it's resource requests in one stage, it blocks the pipeline

O3 CPU Model

`src/cpu/o3/*.hh,cc`

- Detailed out-of-order CPU
  - Default 7-stage pipeline
    - Fetch, Decode, Rename, IEW,Commit
    - IEW <=>Issue, Execute, and Writeback
    - Model varying amount of pipeline stages by changing delays between pipeline stages (e.g. fetchToDecodeDelay)
  - Key Resources
    - Physical Register (PR) File, IQ, LSQ, ROB, Functional Unit (FU) Pool
  - Key Parameters
    - Interstage pipeline delays, Hardware threads, IQ/LSQ/ROB/PR entries, FU Delays
  - Other Key Features
    - Support for CISC decoding (e.g. x86)
    - Renaming with a Physical Register (PR) File
    - Functional units with varying latencies
    - Branch Prediction
    - Memory dependence prediction

Outline

Ruby Memory System

Derek Hower

University of Wisconsin, Madison

Outline

- Feature Overview
- Rich Configuration
- Rapid Prototyping
  - SLICC
  - Modular & Detailed Components
- Lifetime of a Ruby memory request

Feature Overview

- *Flexible* Memory System
  - Rich configuration - Just run it
    - Simulate combinations of caches, coherence, interconnect, etc...
  - Rapid prototyping - Just create it
    - Domain-Specific Language (SLICC) for coherence protocols
    - Modular components
- Detailed statistics
  - e.g., Request size/type distribution, state transition frequencies, etc...
- Detailed component simulation
  - Network (fixed/flexible pipeline and simple)
  - Caches (Pluggable replacement policies)
  - Memory (DDR2)

Rich Configuration - Just run it

- Can build many different memory systems
  - CMPs, SMPs, SCMPs
  - 1/2/3 level caches
  - Pt2Pt/Torus/Mesh Topologies
  - MESI/MOESI coherence
- Each components is individually configurable
  - Build heterogeneous cache architectures (new)
  - Adjust cache sizes, bandwidth, link latencies, etc...
- Get research started without modifying code!

Configuration Examples

1. 8 core CMP, 2-Level, MESI protocol, 32K L1s, 8MB 8-banked L2s, crossbar interconnect
   - `scons build/ALPHA_FS/gem5.opt PROTOCOL=MESI_CMP_directory RUBY=True`
   - `./build/ALPHA_FS/gem5.opt configs/example/ruby_fs.py -n 8 --l1i_size=32kB --l1d_size=32kB --l2_size=8MB --num-l2caches=8 --topology=Crossbar --timing`
2. 64 socket SMP, 2-Level on-chip Caches, MOESI protocol, 32K L1s, 8MB L2 per chip, mesh interconnect
   - `scons build/ALPHA_FS/gem5.opt PROTOCOL=MOESI_CMP_directory RUBY=True`
   - `./build/ALPHA_FS/m5.opt configs/example/ruby_fs.py -n 64 --l1i_size=32kB --l1d_size=32kB --l2_size=512MB --num-l2caches=64 --topology=Mesh --timing`

- Many other configuration options
- Protocols only work with specific architectures (see wiki)

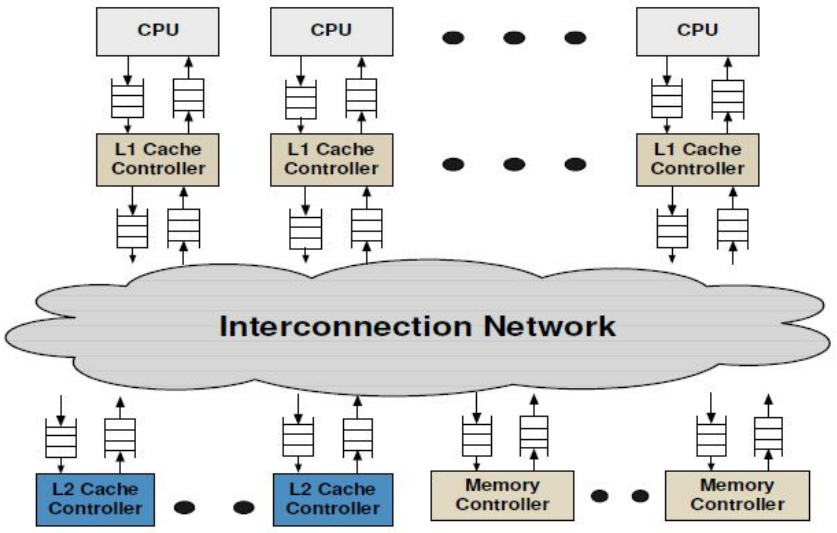
Rapid Prototyping - Just create it

- Modular construction
  - Coherence controller (SLICC)
  - Cache (C++)
    - Replacement Policy (C++)
  - DRAM (C++)
  - Topology (Python)
  - Network implementation (C++)
- Debugging support

SLICC: Specification Language for Implementing Cache Coherence

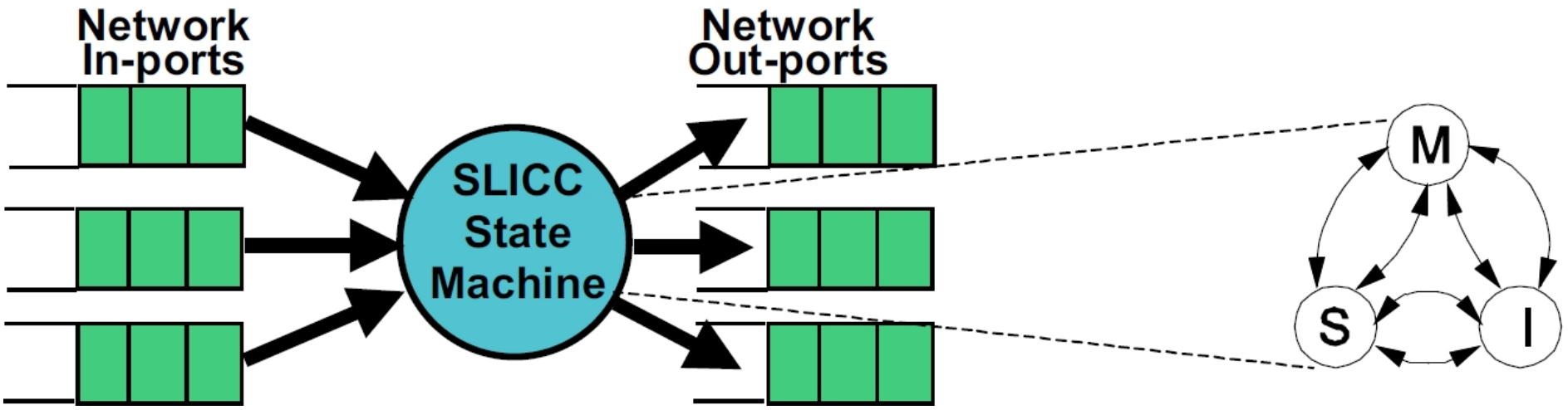
- Domain-Specific Language
  - Syntatically similar to C/C++
  - Like HDLs, constrains operations to be hardware-like (e.g., no loops)
- Two generation targets
  - C++ for simulation
    - Coherence controller object
  - HTML for documentation
    - Table-driven specification (State x Event -> Actions & next state)

SLICC Protocol Structure



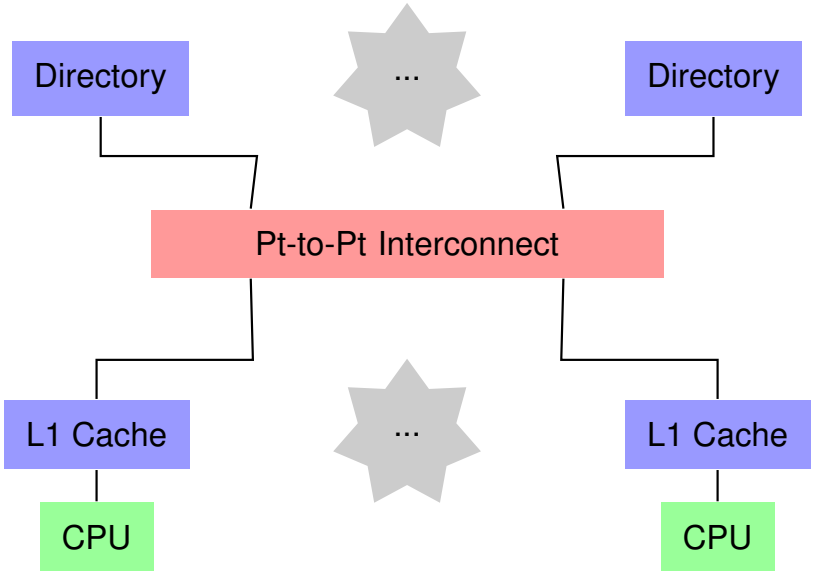
- Collection of *Machines*, e.g.
  - L1 Controller
  - L2 Controller
  - DRAM Controller
- Machines are connected through *network ports* (**different** than MemPorts)
- Network can be an arbitrary topology

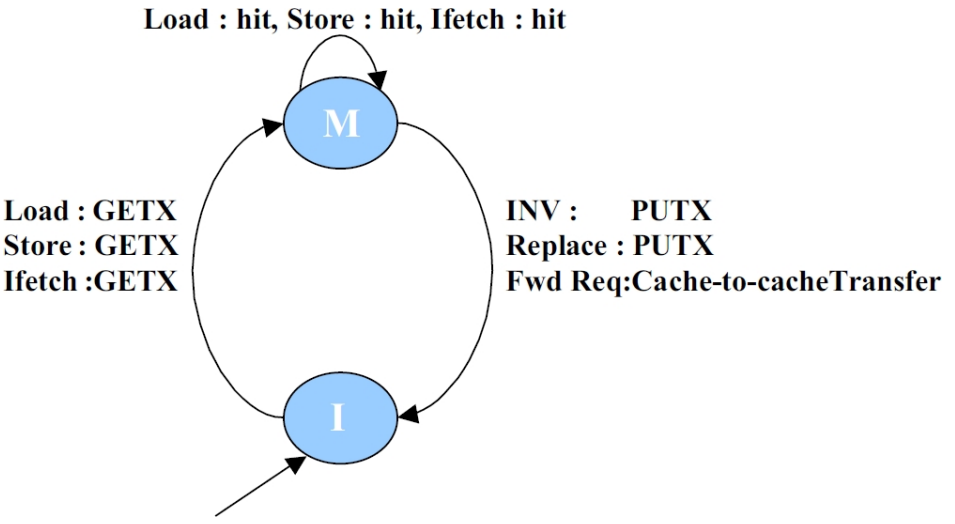
Machine Structure



- Machines are (logically) per-block
- Consist of:
  - **Ports** - Interface to the world
  - **States** - Both stable and transient
  - **Events** - Triggered by incoming messages
  - **Transitions** - Old state x Event -> New state
  - **Actions** - Occur atomically during transition, e.g., Send/receive messages from network

MI Example





- Single-level coherence protocol
  - 2 controller types – Cache + Directory
- Cache Controller
  - 2 stable states: Modified (a.k.a. Valid), Invalid
- Directory Controller [Not Shown]
  - 2 stable states: Modified (Present in cache), Valid
- 3 virtual networks (request, response, forward)
- See `src/mem/ruby/protocols/MI_example.*`

MI Example - L1 Cache Controller

Machine structure

Machine Pseduo-Code

```
machine(L1Cache, "MI Example L1 Cache'')
  : Sequencer * sequencer,                // parameters to the machine object (set at initialization
    CacheMemory * cacheMemory,
    int cache_response_latency = 12,
    int issue_latency = 2
{
   <Define machine interface> // 3 virtual channels to/from network + connection to CPU

   <Declare States & Events>  // M,I,<transients> & Load,Store, etc.

   <Map incoming messages to Events> // e.g., RequestMessage::GETX -> Fwd_GETX

   <Define Actions> // e.g., issueRequest

   <Define Transitions> // e.g., I x Store -> M
}
```

MI Example - L1 Cache Controller

Defining a machine interface

Interface to the network

```
 // MessageBuffers - opaque C++ communication queues
MessageBuffer requestFromCache, network="To", virtual_network="0", ordered="true";
MessageBuffer responseFromCache, network="To", virtual_network="1", ordered="true";
MessageBuffer forwardToCache, network="From", virtual_network="2", ordered="true";
MessageBuffer responseToCache, network="From", virtual_network="1", ordered="true";
```

MI Example - L1 Cache Controller

Defining a machine interface

Interface to the network

```
 // MessageBuffers - opaque C++ communication queues
MessageBuffer requestFromCache, network="To", virtual_network="0", ordered="true";
MessageBuffer responseFromCache, network="To", virtual_network="1", ordered="true";
MessageBuffer forwardToCache, network="From", virtual_network="2", ordered="true";
MessageBuffer responseToCache, network="From", virtual_network="1", ordered="true";

 // out_port - map request type to outgoing message buffer
out_port(requestNetwork_out, RequestMsg, requestFromCache);
out_port(responseNetwork_out, ResponseMsg, responseFromCache);
```

MI Example - L1 Cache Controller

Defining a machine interface

Interface to the network

```
 // MessageBuffers - opaque C++ communication queues
MessageBuffer requestFromCache, network="To", virtual_network="0", ordered="true";
MessageBuffer responseFromCache, network="To", virtual_network="1", ordered="true";
MessageBuffer forwardToCache, network="From", virtual_network="2", ordered="true";
MessageBuffer responseToCache, network="From", virtual_network="1", ordered="true";

 // out_port - map request type to outgoing message buffer
out_port(requestNetwork_out, RequestMsg, requestFromCache);
out_port(responseNetwork_out, ResponseMsg, responseFromCache);

 // in_port - map request type to incomming message buffer
//           and produce code to accept incomming messages
in_port(forwardRequestNetwork_in, RequestMsg, forwardToCache) { ...  }
in_port(responseNetwork_in, ResponseMsg, responseToCache) { ...  }
```

MI Example - L1 Cache Controller

Defining a machine interface

Interface to the network

```
 // MessageBuffers - opaque C++ communication queues
MessageBuffer requestFromCache, network="To", virtual_network="0", ordered="true";
MessageBuffer responseFromCache, network="To", virtual_network="1", ordered="true";
MessageBuffer forwardToCache, network="From", virtual_network="2", ordered="true";
MessageBuffer responseToCache, network="From", virtual_network="1", ordered="true";

 // out_port - map request type to outgoing message buffer
out_port(requestNetwork_out, RequestMsg, requestFromCache);
out_port(responseNetwork_out, ResponseMsg, responseFromCache);

 // in_port - map request type to incomming message buffer
//           and produce code to accept incomming messages
in_port(forwardRequestNetwork_in, RequestMsg, forwardToCache) { ...  }
in_port(responseNetwork_in, ResponseMsg, responseToCache) { ...  }
```

Interface to a CPU

```
 // The other end of mandatoryQueue attaches to Sequencer
MessageBuffer mandatoryQueue, ordered="false";
in_port(mandatoryQueue_in, RubyRequest, mandatoryQueue, desc="...")  { ...  }
// There is no corresponing out_port - handled with hitCallback
```

gems

MI Example - L1 Cache Controller

Declaring States

State Declaration

```
// STATES
 state_declaration(State, desc="Cache states") {
   // Stable States
   I, AccessPermission:Invalid, desc="Not Present/Invalid";
   M, AccessPermission:Read_Write, desc="Modified";

   // Transient States
   II, AccessPermission:Busy, desc="Not Present/Invalid, issued PUT";
   MI, AccessPermission:Busy, desc="Modified, issued PUT";
   MII, AccessPermission:Busy, desc="Modified, issued PUTX, received nack";
   IS, AccessPermission:Busy, desc="Issued request for LOAD/IFETCH";
   IM, AccessPermission:Busy, desc="Issued request for STORE/ATOMIC";
 }
```

MI Example - L1 Cache Controller

Declaring Events

Event Declaration

```
// EVENTS
enumeration(Event, desc="Cache events") {
  // from processor
  Load,           desc="Load request from processor";
  Ifetch,         desc="Ifetch request from processor";
  Store,          desc="Store request from processor";

  // From network (directory)
  Data,           desc="Data from network";
  Fwd_GETX,       desc="Forward from network";
  Inv,            desc="Invalidate request from dir";
  Writeback_Ack,  desc="Ack from the directory for a writeback";
  Writeback_Nack, desc="Nack from the directory for a writeback";

  // Internally generated
  Replacement,    desc="Replace a block";
}
```

MI Example - L1 Cache Controller

Mapping messages to events

- Mapping occurs in `in_port` declaration.
- **`peek(in_port, message_type)`**
  - Sets variable `in_msg` to head of in_port queue.
- **`trigger(Event, address)`**

Event mapping

```
in_port(forwardRequestNetwork_in, RequestMsg, forwardToCache) {
  if (forwardRequestNetwork_in.isReady()) {
    peek(forwardRequestNetwork_in, RequestMsg) {
      if (in_msg.Type == CoherenceRequestType:GETX) {
        trigger(Event:Fwd_GETX, in_msg.Address);
      }
      ...
    }
  }
}
```

gem5

MI Example - L1 Cache Controller

Defining Transitions

- **`transition(Starting State(s), Event, [Ending State]) [ { Actions } ]`**

Transition sequence for new Store request

```
transition(I, Store, IM) {
  v_allocateTBE;            // allocate TBE (a.k.a. MSHR) on transition to transient state
  i_allocateL1CacheBlock;
  a_issueRequest;
  m_popMandatoryQueue;
}

transition(IM, Data, M) {
  u_writeDataToCache;
  s_store_hit;
  w_deallocateTBE;          // deallocate TBE on transition back to stable state
  n_popResponseQueue;
}
...
```

gem5

MI Example - L1 Cache Controller

Defining Actions

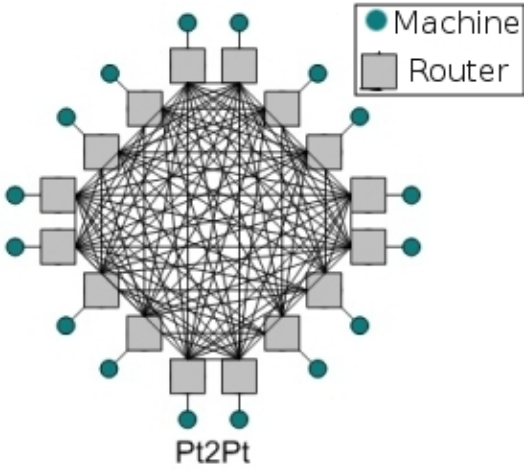
- `action(name, abbrev, [desc]) { implementation }`
- Two special functions available in *action*
  - `peek(in_port, message_type) { use in_msg }`
    - assigns `in_msg` to message at head of port
  - `enqueue(out_port, message_type, [options]) { set out_msg }`
    - enqueues `out_msg` on `out_port`
- Special variable `address` is available inside an action block
  - Set to the address associated with the event that caused the calling transition

Example Action Definition

```
action(e_sendData, "e", desc="Send data from cache to requestor") {
  peek(forwardRequestNetwork_in, RequestMsg) {
    enqueue(responseNetwork_out, ResponseMsg, latency=cache_response_latency) {
      out_msg.Address := address;
      out_msg.Type := CoherenceResponseType:DATA;
      out_msg.Sender := machineID;
      out_msg.Destination.add(in_msg.Requestor);      // uses in_msg set by peek
      out_msg.DataBlk := cacheMemory[address].DataBlk;
      out_msg.MessageSize := MessageSizeType:Response_Data;
147   }
```

MI Example

Connecting SLICC Machines with a Topology



Creating the Topology – Not In SLICC

src/mem/ruby/network/topologies/Pt2Pt.py

```
# returns a SimObject for for a Pt2Pt Topology
def makeTopology(nodes, options, IntLink, ExtLink, Router):
    # Create an individual router for each controller (node),
    # and connect them (ext_links)
    routers = [Router(router_id=i) for i in range(len(nodes))]
    ext_links = [ExtLink(link_id=i, ext_node=n, int_node=routers[i])
                 for (i, n) in enumerate(nodes)]
    link_count = len(nodes)

    # Connect routers all-to-all (int_links)
    int_links = []
    for i in xrange(len(nodes)):
        for j in xrange(len(nodes)):
            if (i != j):
                link_count += 1
                int_links.append(IntLink(link_id=link_count,
                                         node_a=routers[i],
                                         node_b=routers[j]))

    # Return Pt2Pt Topology SimObject
    return Pt2Pt(ext_links=ext_links,
                 int_links=int_links,
                 routers=routers)
```

Using C++ Objects in SLICC

- SLICC can be arbitrarily extended with C++ objects
  - e.g., Interface with a new message filter
- Steps:
  - Create class in C++
  - Declare interface in SLICC with `structure`, external="yes"
  - Initialize object in machine
  - Use!

Extending SLICC

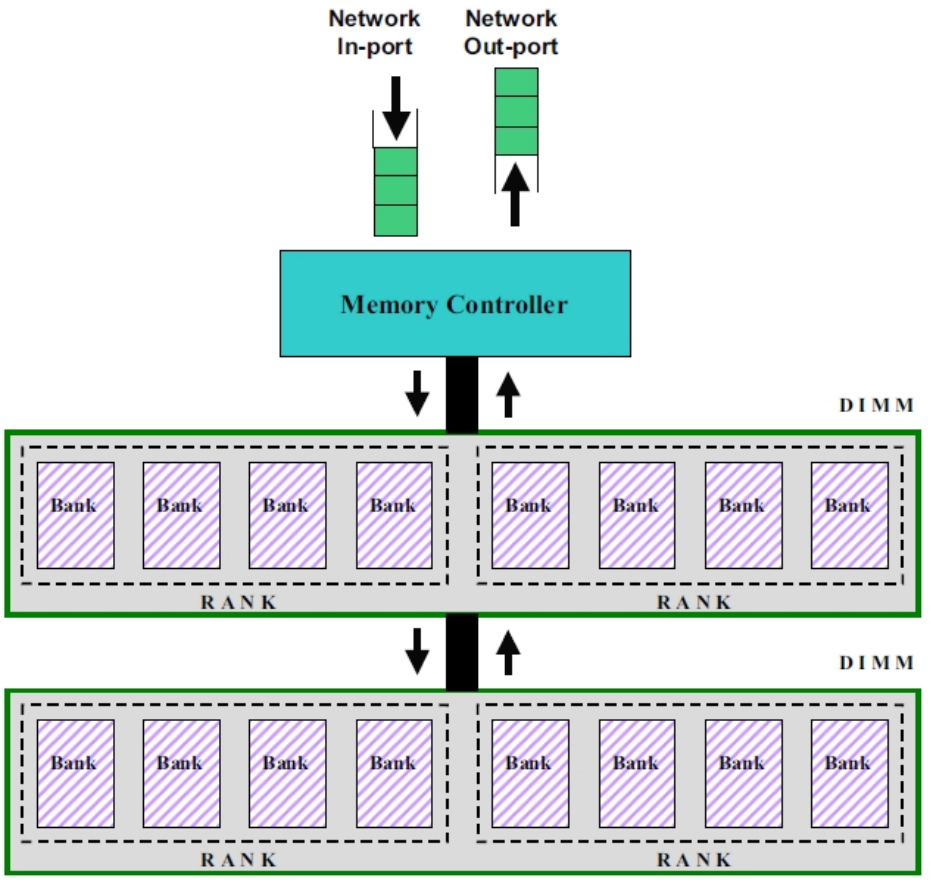
```
// MessageFilter.h
class MessageFilter {
public:
  MessageFilter(int param1);

  // returns 1 if message should be filtered
  int filter(RequestMsg msg);
};

// MessageFilter.cc
int MessageFilter::filter(RequestMsg msg)
{
  ...
  return 0;
}
```

```
// MI_example-cache.sm
structure(MessageFilter, external="yes") {
  int filter(RequestMsg);
};

MessageFilter requestFilter,
   constructor_hack="param";

action(af_allocateUnlessFiltered, "af") {
   if (requestFilter.filter(in_msg) != 1) {
     cacheMemory.allocate(address, new Entry);
   }
}
```

Detailed Component Simulation: Caches

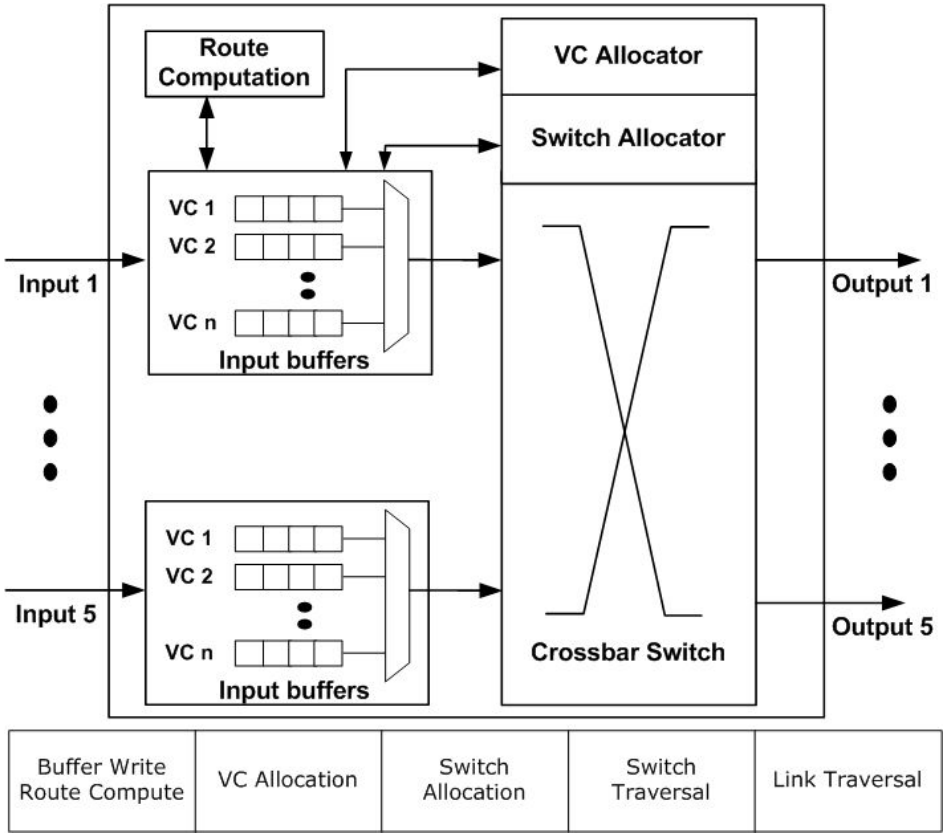
- Set-Associative Caches
- Each CacheMemory object represents one *bank* of cache
- Configurable bit select for indexing
- Modular replacement policy
  - Tree-based pseudo-LRU
  - LRU
- See `src/mem/ruby/system/CacheMemory.hh`

Detailed Component Simulation: Memory



- Memory controller models a **single channel** DDR2 controller
- Implements closed-page policy
- Can configure ranks, tCAS, refresh, etc..
- See `src/mem/ruby/system/MemoryController.hh`

Detailed Component Simulation: Network

- Simple Network
  - Idealized routers - fixed latency, no internal resources
  - **Does** model link bandwidth
- Garnet Network
  - Detailed routers - both fixed and flexible pipeline model
  - From Princeton, MIT
  - 

Ruby Debugging Support

- Random testing support
  - Stresses protocol by inserting random timing delays
- Support for coherence transition tracing
- Frequent assertions
- Deadlock detection

Lifetime of a Ruby Memory Request

1. Request enters through `RubyPort::recvTiming`, is converted to RubyRequest, and passed to *Sequencer*.
2. Request enters SLICC controllers through `Sequencer::makeRequest` via *mandatoryQueue*.
3. Message on mandatoryQueue triggers an event in L1 Controller.
4. Until request is completed:
   1. (Event, State) is matched to a transition.
   2. Actions in the matched transition (optionally) send messages to network & allocate TBE.
   3. Responses from network trigger more events.
5. Last event causes action that calls `Sequencer::hitCallback` & deallocates TBE.
6. RubyRequest is converted back into a Packet & sent to RubyPort.

Outline

Wrap-Up

Brad Beckmann

AMD Research

Summary

- Reviewed the basics
  - High-level features
  - Debugging
  - Checkpointing
- Highlighted new aspects
  - ISA changes: x86 & ARM
  - InOrder CPU model
  - Ruby memory system
- Upcoming Computer Architecture News (CAN) article
  - Summarizes goals, features, and capabilities
  - Please cite if you use gem5
- Overall gem5 has a wide range of capabilities
  - ...but not all combinations currently work

Cross-Product Table

Processor		Memory System		
CPU Model	System Mode	Classic	Ruby	
			Simple	Garnet
Atomic Simple	SE			
	FS			
Timing Simple	SE			
	FS			
InOrder	SE			
	FS			
O3	SE			
	FS			

Spectrum of choices (light = speed, dark = accuracy)

Matrix Examples: Alpha and x86

Alpha				
Processor		Memory System		
CPU Model	System Mode	Classic	Ruby Simple	Ruby Garnet
Atomic Simple	SE FS			
Timing Simple	SE FS			
InOrder	SE FS			
O3	SE FS			

x86				
Processor		Memory System		
CPU Model	System Mode	Classic	Ruby Simple	Ruby Garnet
Atomic Simple	SE FS			
Timing Simple	SE FS			
InOrder	SE FS			
O3	SE FS			

definitely works | should work | might work | definitely does not work

Overall a lot of work remains

- Alpha is more stable than x86
- Ruby does not support atomic accesses
- InOrder model is new
- The O3-Ruby interface is work in progress
- See each ISA status: http://www.gem5.org/Status_Matrix