Using the M5 Simulator

ISCA-33

Steve Reinhardt† Nathan Binkert‡ Ali Saidi
Ron Dreslinski Kevin Lim

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
†Also with Reservoir Labs, Inc.
‡Also with Arbor Networks, Inc.

June 18th, 2006

Welcome!

- We're glad you're here!
- This tutorial is for **you**
  - Please ask questions! Don't save them for the break!
  - We intend the focus to be audience driven

Outline

1. Introduction & Overview
2. Compiling & Running M5
3. Full System Workloads
4. Current M5 Object Models
   - CPU Models
   - Memory System
   - I/O Models
5. Extending M5
6. Wrap-Up

Introduction & Overview

Introduction & Overview

Steve Reinhardt

Introduction & Overview

- What M5 is and is not
- A brief peek inside
- Current status & future developments

What is M5?

- A tool for simulating systems
  - Not just CPU cores: memory, I/O
  - Not just SPEC apps: full OS code
  - Not just single machines: client/server, etc.

Two Views of M5

View #1

- A framework for event-driven simulation
  - Events, objects, statistics, configuration

View #2

- A collection of predefined object models
  - CPUs, caches, busses, devices, etc.

- This tutorial focuses on #2
- You may find #1 useful even if #2 is not

Where Did M5 Come From?

- Born of frustration with existing tools
  - Did not do what we wanted
  - Did not scale with added complexity
- Desire to simulate TCP/IP performance
  - Full-system support with detailed I/O modeling
  - Multiple system simulation
- Almost entirely original code
  - Some SimpleScalar & SimOS used for bootstrapping
  - Now entirely replaced / segregated / deprecated
- No premeditated distribution plans
  - Just hacking together the system *we* wanted

Key M5 Attributes

- Heavily object-oriented
  - Key to modularity, flexibility
- Necessarily complex
  - ∼110K lines of C++, ∼15K lines of Python
- Modular enough to hide the complexity
  - We hope!
- Free! All the code we wrote is open source
  - BSD-style license

What M5 is *Not*

- A hardware design language
  - Higher level for design space exploration, simulation speed
- A restrictive environment
  - Just C++ & Python with an event queue and a bunch of APIs you can choose to ignore
- Finished!
  - Always room for improvement . . .

What We Would Like M5 to Be

- Something that spares you the pain we've been through
- A community resource
  - Modular enough to localize changes
  - Contribute back, and spare others some pain
- A path to reproducible/comparable results
  - A common platform for evaluating ideas
- Let us know how we can help you contribute
  - Public wiki is up at m5.eecs.umich.edu
  - Public repository access coming soon (v2.0)

A Peek Inside

- *Very* brief overview of a few key concepts:
  - Objects
  - Events
  - Modes

A Peek Inside: Objects

- Everything you care about is an object (C++/Python)
- Derived from SimObject base class
  - Common code for creation, configuration parameters, naming, checkpointing, etc.
- Uniform method-based APIs for object types
  - CPUs, caches, memory, etc.
  - Plug-compatibility across implementations
    - Functional vs. detailed CPU
    - Conventional vs. indirect-index cache
- Easy replication: MPs, multiple systems, . . .

A Peek Inside 2: Events

- Standard event queue timing model
  - Global logical time in “ticks”
  - No fixed relation to real time
    - CPU clock cycles (syscall emulation)
    - Picoseconds (full system)
- Objects schedule their own events
  - Flexibility for detail vs. performance tradeoffs
- E.g., a CPU typically schedules event at regular intervals
  - Every cycle or every *n* picoseconds
  - Won’t schedule self if stalled/idle

A Peek Inside 3: Modes

- M5 has two fundamental modes
- Full system (FS)
    - For booting operating systems
    - Models bare hardware, including devices
    - Support address translation (TLB), privileged instructions
- 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

Current Status

- Version 2.0 *almost* ready... that's what we'll cover today
- Two+ CPU models
    - Simple functional
    - New Execute-in-Execute OoO Model (v2.0)
    - (Old SimpleScalar-based OoO – deprecated)
- ISAs supported
    - Alpha: Fully working
    - SPARC & MIPS: Recently added (v2.0)
        - Works now for simple CPU with syscall emulation
- New memory system (v2.0)
    - Uniform connection model based on "ports"
    - Packet-based interface
        - Bus model: split transactions, snooping coherence
        - Enables point-to-point network models (future work)
- Two major cache models: conventional & indirect index

Current Status (cont'd)

- Syscall emulation mode
  - Alpha: Tru64 or Linux
  - SPARC: Linux or Solaris
  - MIPS: Linux
  - Memory-address or SimpleScalar EIO traces
- Full-system mode
  - Compaq "Tsunami"-based Alpha system
    - Boots Linux 2.4 & 2.6, FreeBSD, L4 (Tru64)
  - Ethernet, IDE disk adapters
  - Handful of pre-built benchmarks available
  - SPARC and MIPS full-system support in progress

Short-term To-do/Wish List

- Finish SPARC and MIPS support
  - Syscall emulation completeness / robustness
  - Full-system support
- Finish re-architecting of memory system
  - Support non-bus interconnects, directory coherence
- Fuller Python integration
  - Richer C++/Python interface
  - Move some existing C++ functions into Python
    - Much more flexibility for non-performance-critical parts
- More ISAs: PowerPC, ARM likely candidates
- More full-system benchmarks
- More community involvement: wiki, on-line repository

Outline

1 Introduction & Overview

2 Compiling & Running M5

3 Full System Workloads

4 Current M5 Object Models
   CPU Models
   Memory System
   I/O Models

5 Extending M5

6 Wrap-Up

Compiling & Running M5

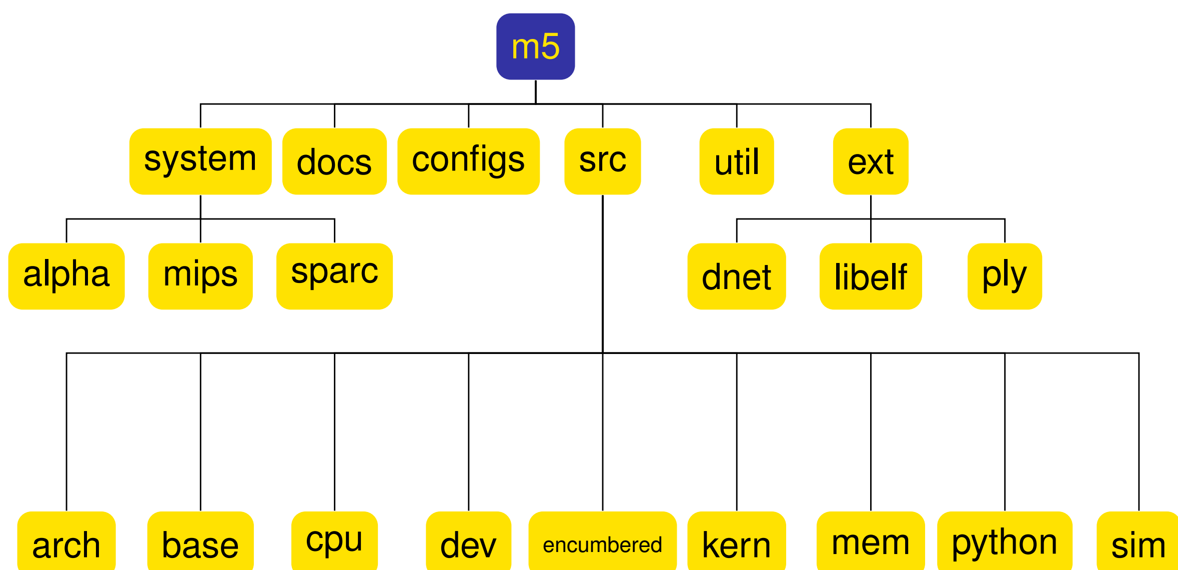
Compiling & Running M5

Nathan Binkert

Compiling & Running M5

- Source tree
- Building executables
- Running simulations
- Output files
- M5 Scripting
- Checkpointing
- Sampling & warm-up

Source Tree Organization



M5 Source Tree Organization

- configs: sample m5 scripts
- src/arch: architecture definition & ISA-specific components
- src/base: general data structures/facilities
- src/python: Python config code
- src/cpu, src/mem, src/dev: specific models
- src/sim: simulator base functionality
- system: platform specific code (palcode, firmware, bios, etc.)
- test: regression tests
- util: utility programs

Building Executables

- Platforms
  - Linux, BSD, MacOS, CYGWIN (most UNIX like systems?)
  - Little endian machines (Big endian support experimental)!
  - 64-bit machines help a lot
- Tools
  - GCC/G++ 3.0+
    - Recently tested with 3.3-3.5,4.0
  - Python 2.4+
  - SCons (bleeding edge 0.96.91)
    - http://www.scons.org
  - SWIG (bleeding edge 1.3.28)
    - http://www.swig.org

Compile Targets

- build/<config>/<binary>
  - configs
    - ALPHA_SE (Alpha syscall emulation)
    - ALPHA_FS (Alpha full system)
    - MIPS_SE (MIPS syscall emulation)
    - SPARC_SE (SPARC syscall emulation)
    - can also define new configs with different compile option settings
  - binary
    - m5.debug – debug build
    - m5.opt – optimized build + debugging symbols + tracing
    - m5.fast – optimized build - no debugging

Sample Compile

```
% scons build/ALPHA_FS/m5.debug

scons: Reading SConscript files ...
Checking for C header file fenv.h... yes
Building in /tmp/newmem/build/ALPHA_FS
Options file /tmp/newmem/build/options/ALPHA_FS not found,
  using defaults in build_opts/ALPHA_FS
Compiling in ALPHA_FS with MySQL support.
scons: done reading SConscript files.
scons: Building targets ...
g++ -o build/ALPHA_FS/base/circlebuf.do -c -pipe -fno-strict-aliasing
    -Wall -Wno-sign-compare -Werror -Wundef -g3 -gdwarf-2 -O0
    -DTHE_ISA=ALPHA_ISA -DDEBUG -Iext/dnet -I/usr/include/python2.4
    -Ibuild/libelf/include -I/usr/include/mysql -Ibuild/ALPHA_FS
    build/ALPHA_FS/base/circlebuf.cc
...
```

Running Simulations

```
Usage:
m5.debug [-p <path>] [-i ] [-h] <script file> [ script options ]

 -p, --path <path>  prepends <path> to PYTHONPATH instead of using
                    built-in zip archive.  Useful when developing/debugging
                    changes to built-in Python libraries, as the new Python
                    can be tested without building a new m5 binary.

 -i, --interactive  forces entry into interactive mode after the supplied
                    script is executed (just like the -i option to  the
                    Python interpreter).

 -h                 Prints this help

 <script file>      script file name which ends in .py. (Normally you can
                    run <script file> --help to get help on that script files'
                    parameters.
```

Sample Run

```
% ./build/ALPHA_FS/m5.debug configs/test/single_fs.py
M5 Simulator System
Copyright (c) 2001-2006
The Regents of The University of Michigan
All Rights Reserved

M5 compiled Jun 13 2006 16:51:37
M5 started Tue Jun 13 16:55:59 2006
M5 executing on zizzer.eecs.umich.edu
      0: system.tsunami.io.rtc: Real-time clock set to Sun Jan  1 00:00:00 2006
Listening for console connection on port 3456
0: system.remote_gdb.listener: listening for remote gdb #0 on port 7000
warn: Entering event queue @ 0.  Starting simulation...
```

Output Files

- Output directory
  - config.out
  - console.<system>.sim_console
  - m5stats.txt
  - cpt.<number>/
- Database output
  - M5 can output stats to a MYSQL database
  - Automatic graph generation (m5/utils/stats)

m5term

- Allows user to connect to the simulated console interface

```
% cd m5
% cd util/term
% make
gcc -o m5term term.c
% make install
sudo install -o root -m 555 m5term /usr /local/bin
```

Sample Terminal Output

```
% m5term localhost 3456
==== m5 slave console: Console 0 ====
M5 console
Got Configuration 127
memsize 8000000 pages 4000
First free page after ROM 0xFFFFFC0000018000
HWRPB 0xFFFFFC0000018000 l1pt 0xFFFFFC0000040000 l2pt 0xFFFFFC0000042000
l3pt_rpb 0xFFFFFC0000044000 l3pt_kernel 0xFFFFFC0000048000 l2reserv
0xFFFFFC0000046000
CPU Clock at 2000 MHz IntrClockFrequency=1024
Booting with 1 processor(s)
...
...
VFS: Mounted root (ext2 filesystem) readonly.
Freeing unused kernel memory: 480k freed
init started: BusyBox v1.00-rc2 (2004.11.18-16:22+0000) multi-call binary

PTXdist-0.7.0 (2004-11-18T11:23:40-0500)

mounting filesystems ...
EXT2-fs warning: checktime reached, running e2fsck is recommended
loading script...
Script from M5 readfile is empty, starting bash shell...
# ls
benchmarks  etc         lib         mnt         sbin        usr
bin         floppy      lost+found  modules     sys         var
dev         home        man         proc        tmp         z
#
```


M5 Scripts

- Python
- Config objs mapped to simulator objs
- No need for scripts to generate multiple configuration files
  - All logic for running many simulations contained in a single configurable script!
- Variables with units are enforced
  - Latency must be '2ns', not simply 2
- Define and parse parameters in the script

```
import os, optparse, sys
import m5
from m5.objects import *

parser = optparse.OptionParser(option_list=m5.standardOptions)
parser.add_option("-t", "--timing", action="store_true")
(options, args) = parser.parse_args()
if args:
    sys.exit('too many arguments')
parser.parse_args()
```

Sample Configuration

```
class System(LinuxSystem):
    if options.timing:
        cpu = DetailedCPU()
    else:
        cpu = SimpleCPU()
    membus = Bus(width=16, clock='400MHz')
    ram = BaseMemory(in_bus=Parent.membus, latency='40ns',
        addr_range=[Parent.physmem.range])
    physmem = PhysicalMemory(range=AddrRange('128MB'))
    tsunami = Tsunami()
    simple_disk = SimpleDisk(disk=Parent.tsunami.disk0.image)
    sim_console = SimConsole(listener=ConsoleListener(port=3456))
    kernel = '/dist/m5/system/binaries/vmlinux-latest'
    pal = '/dist/m5/system/binaries/ts_osfpal'
    console = '/dist/m5/system/binaries/console_ts'
    boot_osflags = 'root=/dev/hda1 console=ttyS0'

root = Root()
root.client = System(readfile='/dist/m5/system/boot/netperf-stream-client.rcS')
root.server = System(readfile='/dist/m5/system/boot/netperf-server.rcS')
root.etherlink = EtherLink(int1=Parent.server.tsunami.etherint[0],
                           int2=Parent.client.tsunami.etherint[0])

m5.instantiate(root) # instantiate structure
code = m5.simulate() # simulate until termination

print 'Exiting @ cycle', m5.curTick(), 'because', code.getCause()
```


Checkpointing

- Creating checkpoints
  - Script commands
  - M5 instruction
    - Insert special instruction into code to trigger a checkpoint to be dropped
    - Our benchmarks do this
- Running checkpoints
  - Same configuration as normal, just add a load checkpoint command to your script.
  - Must regenerate checkpoints with some config changes
    - Most config changes that are architecturally visible (because the kernel may have behaved differently)
    - Physical memory size, new kernels

Sampling & Warm-up

- Sampler object can switch CPU models on the fly
  - E.g., functional ⇒ detailed
  - Statistics dumped & reset at switch
  - Warm-up caches with a functional CPU, do measurements with a detailed CPU
- Can dump and/or reset statistics at other points too
  - E.g., measure statistics across disjoint intervals

Outline

1 Introduction & Overview
2 Compiling & Running M5
3 Full System Workloads
4 Current M5 Object Models
CPU Models
Memory System
I/O Models
5 Extending M5
6 Wrap-Up

Full System Workloads

Full System Workloads

Ali Saidi

Up and Running Benchmarks

- All networking focused
- Surge
- SpecWEB99
- iSCSI
- Netperf
  - stream – a transmit benchmark
  - maerts – a receive benchmark
- In progress:
  - NFS (server works; client tuning needed)

Two Parts

- Disk images
  - Complete image (partition table and partition image in one)
  - Binaries to be run must be present on image
- rcS files (**`m5/configs/boot/*.rcS`**)
  - Exactly like normal boot scripts
  - Nice and flexible, since not compiled in
    - Specified in configuration by **`readfile='path/to/script.rcS'`**
  - Use them to start running a binary on the disk image, configure ethernet interfaces, execute m5 instructions, etc.

See for yourself!

Going into / of disk image and typing ls will show:

```
benchmarks  etc     lib         mnt      sbin  usr
bin         floppy  lost+found  modules  sys   var
dev         home    man         proc     tmp   z
```

Snippet of .rcS file:

```
echo -n "setting up network..."
 /sbin/ifconfig  eth0 192.168.0.10  txqueuelen  1000
 /sbin/ifconfig  lo 127.0.0.1
echo -n "running surge client..."
/bin/bash -c " cd  /benchmarks/surge && ./Surge 2 100 1 192.168.0.1 5"
echo -n "halting machine"
m5 exit
```

Adding Your Own Benchmarks

- Full system simulation currently supports Alpha
  - Need to compile Alpha binaries (www.kegel.com/crosstool)
  - If you have an Alpha, use that
- Add the benchmark binaries to disk image
- Create .rcS file that executes the binary
- Please share them with others!

Mounting Disk Images

- If you need to mount a disk image to change something (like add a benchmark binary)

As root:

```
mount -oloop,offset=32256 myimage.img /mnt/point
```

- You can then manipulate the file system directly and copy in binaries
- Don't forget to unmount!

Example

- New benchmark: mybench
- Compile and put it in disk image:
  - **cp mybench /mnt/point/benchmarks/mybench**
- Create .rcS files:

```
#!/bin/sh
ifconfig eth0 192.168.0.1
echo "executing mybench"
eval /benchmarks/mybench
```

Outline

1. Introduction & Overview
2. Compiling & Running M5
3. Full System Workloads
4. Current M5 Object Models
   - CPU Models
   - Memory System
   - I/O Models
5. Extending M5
6. Wrap-Up

CPU Models

CPU Models

Kevin Lim

CPU Section Overview

- Models:
  - Simple CPU
  - O3 CPU
    - Checker
- Key classes:
  - StaticInst – Decoded instruction
  - DynInst – Stores dynamic information
- Architected state:
  - Interface:
    - ThreadContext – External interface
    - ExecContext – ISA Interface
  - Implementation:
    - SimpleThread

Simple CPU Model `src/cpu/simple/base.{hh,cc}`

- Rewritten for new memory system
- Uses of the SimpleCPU:
  - Warming up caches
  - Driving systems that do not require detailed modeling
- Ideal starting point to learn how CPU models work within M5
  - Simple overview of fetching, executing, and retiring instructions
  - Handles all the calls to support full system mode

Simple CPU Subclasses

- AtomicSimpleCPU
  - Memory accesses are atomic
  - Latency of cache accesses are an estimate
  - Fastest functional simulation
- TimingSimpleCPU
  - Memory accesses use timing path
  - CPU waits until memory access returns
  - Fast, provides some level of timing

Simple CPU Details

- Simulates an in-order 1 CPI machine
  - Single threaded
  - Can roughly model a superscalar machine by ticking multiple times
- Can be extended to simple pipeline
  - Make function calls into pipeline stages
  - Add in timing between pipeline stages
  - Add in functional units with latencies

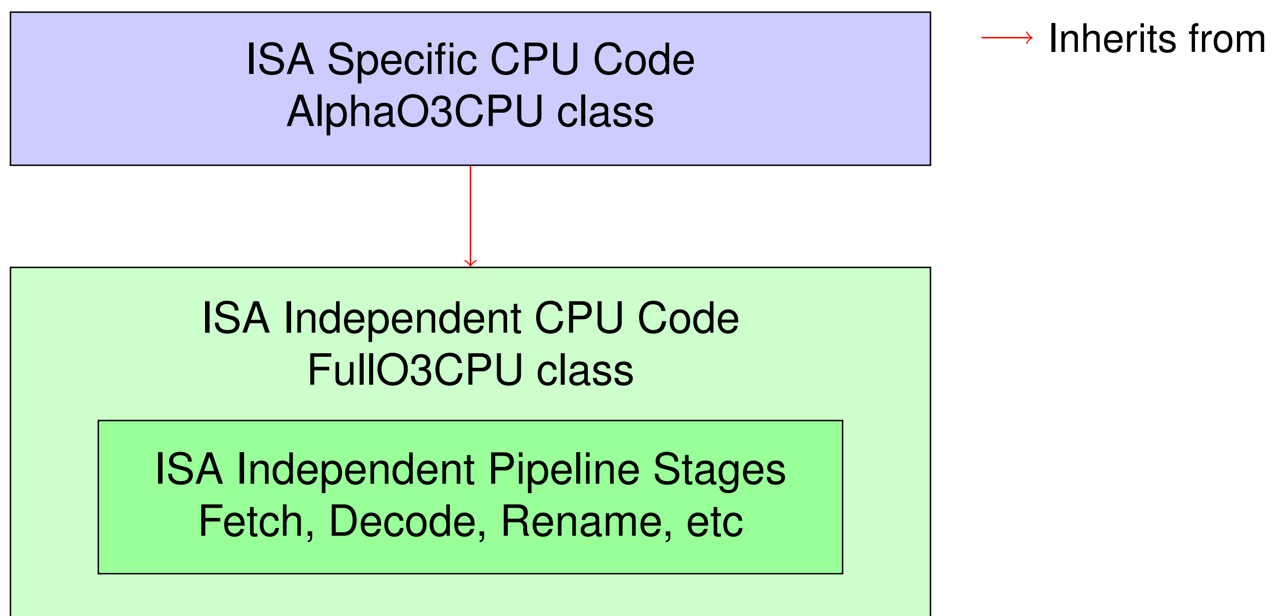
Functionality

- Execution driven
  - Tied closely with SimpleThread class
- Provides ExecContext interface
  - ISA accesses state through CPU
- Works in full system and syscall emulation modes
  - Handles interrupts, traps, etc.

O3 CPU Model `src/cpu/o3/*`

- Detailed out-of-order CPU
  - IQ, ROB, LSQ
  - Renaming with a physical register file
  - Functional units
  - Branch prediction
  - Memory dependence prediction
- Syscall emulation and full-system modes
  - SMT in syscall emulation (for now)

Example

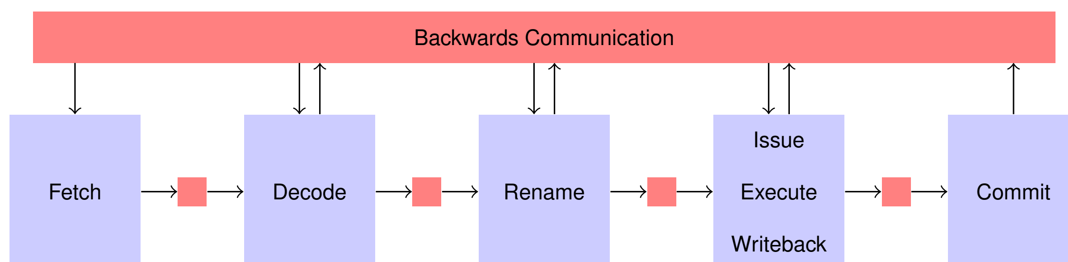


Timing accuracy

- Many models execute instructions at the beginning or end of a pipeline
  - Trades-off timing interaction for other details
- O3 CPU executes at execute, modeling the timing for each pipeline stage
  - Important for coherence
  - UP and MP system studies
- Forces both timing and execution to be accurate

Pipeline layout

- OoO pipeline, loosely based on Alpha 21264
- Low level structure:
  - Red is a time buffer



Time Buffers `src/base/timebuf.hh`

- Similar to queues
  - Are **`advance()`**'d each CPU cycle
- Each pipeline stage places information into time buffer
  - Next stage reads from time buffer by indexing into appropriate cycle
- Used for both forwards and backwards communication

Time Buffer Use

- Time buffer class is templated
  - Its template parameter is the communication struct between stages
- Stages must communicate to each other via the time buffer
  - Avoids unrealistic interaction between pipeline stages

Template Policies

- Template policy classes used to define CPU policies
  - Gives full type information
  - Avoids virtual functions
- “Impl” class is passed in as template parameter to all classes
  - Impl defines all the important types, classes, pipeline stages, etc.

Template Policy Example

- Template policy code example:

```
template<class Impl>
class  DefaultFetch  {
    typedef typename Impl::BP  BranchPred;
    ...
};
```

- DefaultFetch is able to obtain full type of the Branch Predictor
  - Can extend to CPU stages, CPU pointer, instruction types, etc.

Checker

`src/cpu/checker/*`

- Dynamically verifies O3 CPU
  - Supports any CPU that uses DynInst
  - Checks instructions as they complete
- Checks for correct:
  - Path
  - Fetched instructions
  - Execution results

Checker Limitations

- Cannot verify:
  - Uncached accesses
    - Devices do not support speculative reads
    - Device access may affect device state
  - Store conditional results
  - Memory value of loads
    - Memory value may have changed between execute and commit
- Assumes interrupts are handled correctly by the main CPU

O3 CPU Future Directions

- Support for multiple ISA's
  - Currently Alpha is the only one supported
  - Lower levels of code can be shared across platforms
- SMT in full system mode

StaticInst Class `src/cpu/static_inst.{hh,cc}`

- Represents a decoded instruction
  - Has classifications of the inst
  - Corresponds to the binary machine inst
  - Decoded once
  - Only has static information
- Has all the methods needed to execute an instruction
  - Tells which regs are source and dest
  - Contains the **`execute()`** function
  - ISA parser generates **`execute()`** for all insts

DynInst Class `src/cpu/base_dyn_inst.{hh,cc}`

- Dynamic version of StaticInst
  - Used for detailed CPU model
  - Holds PC, results, renamed regs, etc.
- Templated on CPU policy
- Provides ExecContext Interface

Architected state

- Two main interfaces for architected state
  - ThreadContext
  - ExecContext
- Implementation of state and interface:
  - SimpleThread

ThreadContext `src/cpu/thread_context.hh`

- Interface for accessing total architectural state of a single thread
  - PC, register values, etc.
- Used to obtain pointers to key classes
  - Memory ports, process, system, etc.
- Abstract base class
  - Each CPU model must implement its own derived ThreadContext

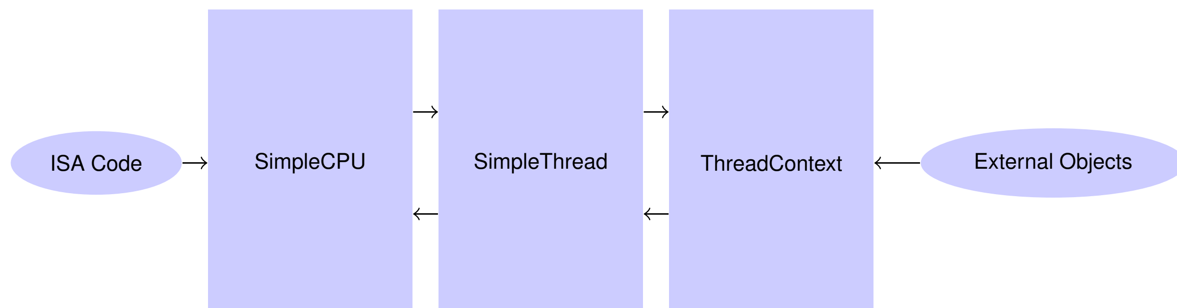
ExecContext `src/cpu/exec_context.hh`

- Implicit interface used by ISA code to access CPU state
- ExecContext is not derived from
  - File is for documentation only
- Implementations:
  - SimpleCPU class
  - DynInst class

SimpleThread `src/cpu/simple_thread.hh`

- Provides all state necessary for execution
    - Holds all architectural state
    - Sufficient for simple CPU models
- Provides same functions as ThreadContext interface
    - Allows for a ThreadContext proxy to call functions on SimpleThread
- Used to be ExecContext in M5 v1.x

Interface Example



Memory System

Memory System

Ron Dreslinski

Outline

- New Memory System Goals
- Requests and Packets
- **`MemObject`**s
- Ports
- Access Modes
- Interconnects
- Caches
- Memory Models

New Memory System Goals

- Combine timing and functional access into one
  - Needed for execute-in-execute CPU model
  - Prevents components from cheating
- Simplify code
  - Remove large pieces of duplicate code (e.g. old bus model)
  - Minimize the amount of templates within the code
- Make the code more easily extensible
  - More modular component connections
  - Enables creation of other interconnects

Requests and Packets `src/mem/(request|packet).*`

- Requests
  - Span entire memory access
  - Contains data persistent throughout the transaction
    - Virtual/Physical Address
    - Size
    - PC, CPU, and Thread ID of initiator
- Packets
  - Encapsulates transfer between two memory objects
    - Address/Size
    - Pointers to the data and request
    - Status condition
    - Command attributes
    - SenderState / CoherenceState (opaque pointers)
  - Accessor functions protecting use of data elements

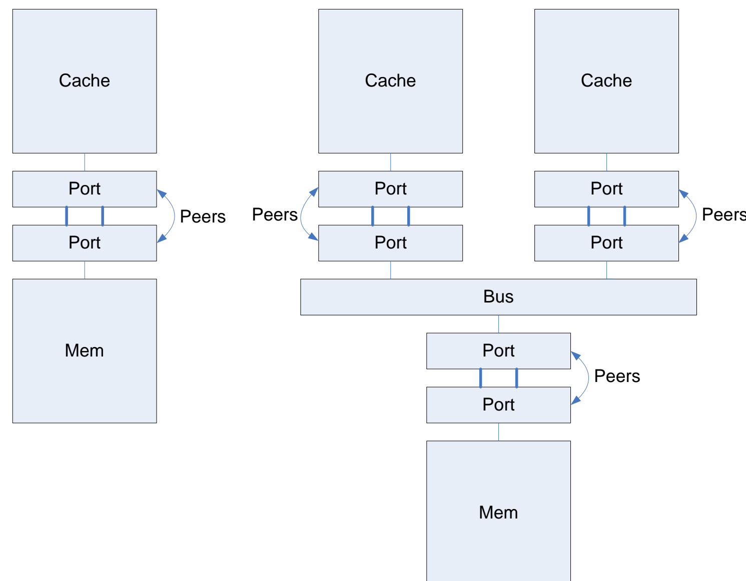
MemObjects `src/mem/mem_object.*`

- All Objects in memory system derive from a base class called **`MemObject`**
  - **`MemObject`** is **`SimObject`** with an additional function exported
  - **`getPort(string name)`** - returns a port with a matching name
  - Connection in memory system made by calling **`getPort()`** on two objects followed by **`setPeer()`** on the resulting ports

Ports `src/mem/port.{hh,cc}`

- New form of connecting **`MemObject`**s together
- Ports come in pairs and the other port is known as the peer
- Two groups of functions in each port object
  - **`send*`** functions are called by the **`MemObject`** to contact the **`peer`** port
  - **`recv*`** are called on the **`peer`** port object by the **`send*`**
- These simple functions keep the interface generic

Ports Example



Current M5 Object Models | Memory System

Access Modes – Functional

- Accesses happen instantaneously updating data everywhere in the hierarchy
- Used for loading binaries, remote debuggers, etc.
- If devices contain queues of packets they must be scanned and updated as well

Access Modes – Atomic & Timing

- Atomic
  - Return approximate time to complete request without resource contention or queing delay
  - Used for fast forwarding or warming caches
- Timing
  - Most detailed access
  - Models all timing/queuing in the memory system
  - Once a timing request is sent successfully it will later receive a response or a NACK
- Atomic and Timing accesses can not co-exist at the same time

Interconnects

- Interconnect models simply help route packets from one device to another
- With the generic port interface any number of interconnects can be defined
- Current support for a simple bus model and bus bridge

Buses and Bridges `src/mem/(bus|bridge).{hh,cc}`

- Buses
  - Configurable width and speed
- Bridges
  - Simple objects to connect two different buses
  - Queues requests coming from either side and forwards them out the other side
  - Configurable amount of queuing space

Caches `src/mem/cache/*`

- Templated on:
  - Cache tags (LRU, etc.)
  - Buffer type (blocking, MSHR)
  - Coherence protocol (uni, MSI, etc.)

Main Cache Parameters

- Size
- Associativity
- Block Size
- Latency
- Trace address

Cache Tags `src/mem/cache/tags/*`

- Holds blocks and block state
- Contains the replacement policy
  - LRU
  - Generational (IIC)

Miss Queue `src/mem/cache/miss/*`

- Blocking buffer
  - Used to simulate a blocking cache
- MSHR
  - Blocks when miss or writeback queue is full
  - Configurable parameters
    - Size of miss and writeback queues
    - Number of targets per MSHR

Coherence Support `src/mem/cache/coherence/*`

- Uni coherence model
  - Single Processor
  - Handles DMA invalidate forwarding up the cache hierarchy
    - via CSHR's
- MOESI based model
  - Also has CSHR support invalidates

MOESI Bus Coherence

- Configurable parameters:
  - Protocol type (MSI,MOSI,MESI,MOESI)
  - Upgrades
- Support for memory to snarf updates
  - Only works when the memory module is connected to the coherent bus

Prefetching `src/mem/cache/prefetcher/*`

- New prefetching models have been added for hardware prefetching
  - Priority given to DemandMisses, separate prefetch queue
  - Several Prefetchers supported
    - N-Block Ahead
    - Tagged Prefetching
    - Stride Prefetching
    - Some variants of Global History Buffer prefetching

Memory Models

- Two main models
  - A simple memory bank with fixed latency
  - A more detailed DRAM model designed with configurable latencies

I/O Models

I/O Models

Ali Saidi

Overview

- Device Basics
- Miscellaneous Devices
- Disk Model
- Network Model

Device Basics `src/dev/*`

- All based on **`MemObject`**
  - **PioDevice** - Acts only as a target for requests with **`pioPort`**
  - **DMADevice** - Can also act as a master with a **`dmaPort`**
  - **PCIDev** - PCI Configuration space, PCI interrupt handling
- Each device responds to one or more address ranges
- A device implements a **`read()`** and **`write()`** for basic PIO reads and writes
- **`DMADevice::dmaWrite(), dmaRead()`** for DMA reads/writes

Miscellaneous Devices

- **AlphaConsole** - Back door into simulator for console code
- **BadDev** - Device panics on access
- **CowDiskImage** - Copy-on-write disk image
- **PciConfigAll** - PCI configuration space object
- **PciDev** - PCI device base class
- **PacketFifo** - FIFO for network packets
- **SimConsole** - Device that provides the console
- **Tsunami** - The platform that links together all it's devices
- **TsunamiCChip** - Tsunami interrupt controller
- **TsunamiPChip** - Tsunami PCI interface
- **TsunamiIO** - Legacy I/O devices (RTC, PIT, etc)
- **Uart** - Serial UART

PCI & Interrupts

- Tsunami platform
- Four physical PCI slots
- Only implement one PCI bus
  - Possible to implement second, just not done
- Devices need their own interrupt and device ID
  - No interrupt sharing allowed
  - Simulator will panic if detected
  - Set `InterruptLine` parameter of PciConfigData to change

Disk Interface

`dev/ide_*.{hh,cc}`

- Modeled as an IDE controller and separate IDE Disk(s)
- Copy-on-write layer in between
- Simple timing support
- Emulates an Intel IDE 2 channel controller
    - Can connect 4 IDE devices

Disk Images

- Can be created with **`mkblankimage.sh`**
- Disk images are contiguous blocks created with dd
- Unlike a normal image it needs to contain a partition table
- Can be mounted with the loopback device

Specifying Disk Image

```
class LinuxRootDisk(IdeDisk):
    raw_image = RawDiskImage(image_file=disk('linux.img'), read_only=True)
    image = CowDiskImage(child=Parent.raw_image, read_only=False)


class LinuxTsunami(BaseTsunami):
    disk0 = LinuxRootDisk(driveID='master')
    disk2 = LinuxSwapDisk(driveID='master')
    ide = IdeController(disks=[Parent.disk0, Parent.disk2],
                        configdata=IdeControllerPciData(),
                        pci_func=0, pci_dev=0, pci_bus=0)
```

NIC Device Model `src/dev/ns_gige.{hh,cc}`

- National Semiconductor DP83820
  - Gigabit Ethernet Full Duplex PCI controller
  - Their spec is actually public
- Modeled Device Features/Components:
  - PCI bus interface
  - Device registers
  - Tx/Rx FIFOs
  - Buffer Management Scheme
  - Receive Packet Filtering Logic
  - Checksum Offloading

NIC Device Model cont.

- Unmodified Linux driver will run on our model (linux/drivers/net/ns83820.c)
  - Actual hardware only allows receive engine to DMA to 8 byte boundaries
  - Fixed in our model and patched kernel
- If Linux doesn't use it, we don't model it
  - Could be added if desired
- Added features
  - Interrupt coalescing – collects interrupts for **`intr_delay`** before interrupting CPU
  - Rx/Tx FIFO sizes – modifiable by setting: **`rx_fifo_size`**, **`tx_fifo_size`**

Etherlink `m5/dev/etherlink.{hh,cc}`

- Configurable
  - Link delay
  - Bandwidth
- Point-to-point
  - Connect any two NICs
- Packet dump
  - Dumps a pcap formatted Ethernet trace
    - Read with tcpdump, Ethereal

Outline

1. Introduction & Overview
2. Compiling & Running M5
3. Full System Workloads
4. Current M5 Object Models
   - CPU Models
   - Memory System
   - I/O Models
5. Extending M5
6. Wrap-Up

Extending M5

Extending M5

Steve Reinhardt & Nate Binkert

Extending M5

• Overview of M5 internals	Steve
• Defining new objects	Steve
• ISA Description Language	Steve
• Debugging	Nate
• Statistics	Nate

Execution Process

`src/sim/main.cc, src/python/m5/__init__.py`

- M5 embeds Python interpreter, simply invokes your script
- **`m5`** Python module exposes M5 functions
  - Native Python via embedded zip archive
  - C++ via SWIG wrappers
- Up to script to:
  - Process command-line args
    - Easy to include standard M5 args (using **`optparse`**)
    - Add your own customized for your simulation
  - Build configuration
  - Unserialize from checkpoint, if any
  - Start processing events from event queue
- On exiting Python, M5 will dump statistics

Building Configurations

- Configuration is a tree of objects
- Build it in Python
  - Instantiate Python SimObject classes
    - “Instantiates” children too... tree copy
  - Can subclass Python SimObjects
    - Override default parameter values
    - Attach child objects
  - Can also “clone” SimObject instances
  - Parameter values are inherited from base class and/or clone ancestor unless overridden
    - Even if base class value is changed later
    - Exception: can’t modify child structure after subclass/clone
- Instantiate C++ copy via **`m5.instantiate(root)`**

Creating New SimObjects

- Derive C++ class from C++ SimObject
  - Defines simulation behavior
  - See **`src/sim/sim_object.{cc,hh}`**
  - Add C++ filename to **`src/SConscript`**
- Derive Python class from Python SimObject
  - Defines parameters, ports for configuration
  - Add file in **`src/python/m5/objects`** directory
  - Add name to **`__init__.py`** in that dir
- C++ needs parameter/creation boilerplate
  - Ugly macros in **`src/sim/builder.hh`**
  - Plan to be cleaning this up soon. . .
- That’s it! Recompile and use!
  - Linking .o file automatically registers object

Serialization `src/sim/serialize.{hh,cc}`

- Create/restore state checkpoints
- Serializable is base of Event & SimObject
  - defines **`serialize()`**, **`unserialize()`** methods
  - override to save/restore object state
  - .ini-format text file
- If checkpoint is specified, M5 will call **`unserialize()`** on all objects after creation
  - State identified by object name (system0.cpu0)
  - OK if no checkpointed state (e.g., added cache)
- Common error: adding field to object and not updating **`serialize()`**/**`unserialize()`**

Events `src/sim/eventq.{hh,cc}`

- Event object is abstract superclass
- Derive new subclass for specific event
  - Add fields for event-specific data
  - Override **`process()`** method for action
- **`schedule(Tick t)`** puts on event queue
- Events may be statically or dynamically allocated
  - Setting AutoDelete flag will call delete after processing

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

Definitions etc.

```
def  bitfield  OPCODE  <31:26>;
def  bitfield  RA  <25:21>;
def  bitfield  RB  <20:16>;
def  bitfield  INTFUNC  <11: 5>; // function code
def  bitfield  RC  < 4: 0>; //  dest   reg

def operands {{
    'Ra': ('IntReg', 'uq', 'PALMODE ? AlphaISA::reg_redir[RA] : RA',
           'IsInteger', 1),
    'Rb': ('IntReg', 'uq', 'PALMODE ? AlphaISA::reg_redir[RB] : RB',
           'IsInteger', 2),
    'Rc': ('IntReg', 'uq', 'PALMODE ? AlphaISA::reg_redir[RC] : RC',
           'IsInteger', 3),
    'Fa': ('FloatReg', 'df', 'FA', 'IsFloating', 1),
    'Fb': ('FloatReg', 'df', 'FB', 'IsFloating', 2),
    'Fc': ('FloatReg', 'df', 'FC', 'IsFloating', 3),
}}

def format  LoadAddress(code) {{
  // Python code here...
}}

def format  IntegerOperate(code) {{
   // Python code here...
}}
```

Instruction Decode & Semantics

```
decode OPCODE {{
   format  LoadAddress {{
    0x08:  lda ({{ Ra =  Rb  +  disp ; }});
    0x09:  ldah ({{ Ra =  Rb  + ( disp  << 16); }});
  }
  format  IntegerOperate {
    0x10: decode INTFUNC {
        0x00:  addl ({{  Rc.sl  =  Ra.sl  +  Rb_or_imm.sl ; }});
        0x20:  addq ({{  Rc  = Ra +  Rb_or_imm ; }});
        0x22: s4addq({{  Rc  = (Ra << 2) +  Rb_or_imm ; }});
        0x32: s8addq({{  Rc  = (Ra << 3) +  Rb_or_imm ; }});
        // etc.
    }
  }
  // etc.
}}
```

Key Features

- Very compact representation
  - Most instructions take 1 line of C code
  - Alpha: 3437 lines of isa description → 39K lines of C++
    - 15K generic decode, 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

Debugging M5

- Tracing with DPRINTF
- Instruction Tracing
- Using the Debugger
- Remote Debugging

Tracing

src/base/trace(flags).*

- printf is a nice debugging tool
- Keep good printfs for tracing
- Lots of debug output is a very good thing
- Add new flags to traceflags.py
  - Individual flags in the baseFlags array
  - Groups of flags in the compoundFlagMap dict
- Fetch, Decode, Ethernet, IPI, TLB, DMA, Bus, Cache, Loader, AlphaConsole, etc.

```
DPRINTF(Flag, "normal printf %s\n", "arguments");

Command line flags:
m5.opt --traceflags="Space Separated List"

From gdb:
(gdb) call setTraceFlag("Flag")
(gdb) call clearTraceFlag("Flag")
```

Instruction Tracing

- Separate from the general trace facility
  - Flags for printing cycle, symbols, etc.
- rundiff
  - Compare two simulations
  - Good for finding bugs

Using GDB with M5

```
% gdb ~/build/newmem/build/ALPHA_FS/m5.debug
GNU gdb 6.1-20040303 (Apple version gdb-437) (Fri Jan 13 18:45:48 GMT 2006)
... more gdb header ...

(gdb) b main
Breakpoint 1 at 0x39027: file /Users/nate/build/newmem/build/ALPHA_FS/sim/main.c, line 175.

(gdb) run configs/test/single_fs.py
Starting program: /Volumes/work/build/newmem/build/ALPHA_FS/m5.debug configs/test/single_fs.py

Breakpoint 1, main (argc=34608072, argv=0x21013e0) at /Users/nate/build/newmem/build/ALPHA_FS/
sim/main.cc:175
175         myProgName = argv[0];
(gdb) call sched_break_cycle(3000)

(gdb) cont
Continuing.

... M5 Output ...

Program received signal SIGTRAP, Trace/breakpoint trap.
0x9003d9ec in kill ()
(gdb) p curTick
$1 = 3000

(gdb)
```

Remote Debugging

```
% ~/m5/build/ALPHA_FS/m5.debug ~/m5/configs/test/single_fs.py
M5 Simulator System
Copyright (c) 2001-2006
The Regents of The University of Michigan
All Rights Reserved

M5 compiled Jun 17 2006 09:58:47
M5 executing on ziff.eecs.umich.edu
M5 started Sat Jun 17 15:15:58 2006
Listening for console connection on port 3456
      0: system.tsunami.io: Real-time clock set to Sun Jan 1 00:00:00 2006
command line: /n/ziff/z/binkertn/build/head/ALPHA_FS/m5.debug
              /n/ziff/z/binkertn/research/m5/head/configs/test/single_fs.py

Listening for remote gdb connection on port 7000
warn: Entering event queue. Starting simulation...
```

Extending M5

Remote Debugging

```
% gdb-linux-alpha  arch/alpha/boot/vmlinux
... gdb banner ...
This GDB was configured as "--host=i686-pc-linux-gnu --target=alpha-linux"...
        (no debugging symbols found)...
(gdb) set remote Z-packet on [This can be put in .gdbinit]
(gdb) target remote ziff:7000
Remote debugging using ziff:7000
0xfffffc0000496844 in strcasecmp (a=0xfffffc0000b13a80 "", b=0x0) at
         arch/alpha/lib/strcasecmp.c:23
23       } while (ca == cb && ca != '\0');
(gdb)
```

The M5 Statistics Package

- Statistics types
  - Scalar
  - Average
  - Vector
  - Formula
  - Distribution
  - Vector Distribution
- M5 has phases, once it moves to the running phase, no new stats

Statistics Example – header file

```
class MySimObject : public SimObject
{
  private:
     Stats::Scalar txBytes;
     Stats::Formula txBandwidth;
     Stats::Vector syscall;

  public:
    void regStats();
};
```

Statistics Example – cc file

```
txBytes
  .name(name() + ".txBytes")
  .desc("Bytes Transmitted")
  .prereq(txBytes)
  ;

txBandwidth
  .name(name() + ".txBandwidth")
  .desc("Transmit Bandwidth (bits/s)")
  .precision(0)
  ;

txBandwidth = txBytes * Stats::constant(8) / simSeconds;

syscall
  .init(SystemCalls <Linux>::Number)
  .name(name() + ".syscall")
  .desc("number of syscalls executed")
  .flags(total | pdf | nozero | nonan)
  ;
```

Statistics Output

```
client.tsunami.etherdev.txBandwidth  4302720
client.tsunami.etherdev.txBytes      13446
server.tsunami.etherdev.txBandwidth  4684921600
server.tsunami.etherdev.txBytes      14640380
sim_seconds                          0.025000
server.cpu.kern.syscall      492
server.cpu.kern.syscall_1    189  38.41%      38.41%
server.cpu.kern.syscall_2    249  50.61%      89.02%
server.cpu.kern.syscall_3     54  10.98%      100.00%
```

Outline

1. Introduction & Overview
2. Compiling & Running M5
3. Full System Workloads
4. Current M5 Object Models
   - CPU Models
   - Memory System
   - I/O Models
5. Extending M5
6. Wrap-Up

Wrap-Up

Wrap-Up

Steve Reinhardt

Thank You!

- We hope you found this tutorial useful
- We hope you find M5 useful too
- We'd love to work with you to make M5 even more useful to the community
- We value your feedback
  - Please fill out a questionnaire
  - Hand it to one of us on your way out, or during ISCA

Keep In Touch

- Come talk to us at ISCA
  - We're all here through Wednesday
- Check http://m5.eecs.umich.edu
  - Use the wiki to learn more
  - Contribute things you learn on your own
- Use, subscribe to our mailing lists:
  - m5sim-users@lists.sourceforge.net
  - m5sim-announce@lists.sourceforge.net
- Look for our article in the July/August issue of IEEE Micro: "The M5 Simulator: Modeling Networked Systems"