Introduction to SESC Simulator

Jie Chen
jiec@gwu.edu
Outline

- Introduction to Simulator
- Environmental Settings
- Building SESC executable
- Running benchmark apps
- SESC code structure
Outline

- Introduction to Simulator
  - Environmental Settings
  - Building SESC executable
  - Running benchmark apps
  - SESC code structure
Why Do We Need a Microprocessor Simulator?

- **When building a new REAL microprocessor, you need**
  - Large teams of experts and supported fund
  - Architecting the functionality of the chip
  - Front end logic design and implementation
  - Back end synthesis, place and route
  - Exhaustive verification and testing
  - Expensive fabrication

- **What can we do if we want to evaluate a new design?**
  - Researchers use microprocessor simulators
    - They are written in software
    - They can run applications
      - like real processors, but slower
    - Using them to evaluate new designs becomes much easier
      - changing configurations
      - adding new components
What is SESC?

- **SuperESCalar Simulator**
  - Developed primary by i-acoma group at UIUC
  - Widely used in Academia

- **Microprocessor architectural simulator**
  - MIPS instruction set
  - Uniprocessors
  - Chip Multi-processors (CMP)

- **Implemented in software**
  - Open source, available at Sourceforge website
  - Modularized source code structures
  - Written in C++ and high optimized for speed

- **Event-Driven Simulation**
  - Function simulation (done by emulation part)
  - Timing simulation (done by the rest parts)
Documentation

- High level explanation of SESC
  - http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/

- README files in SESC source package
  - sesc/docs

- SESC source code
  - The best documentation

- Google “sesc simulator” online
Outline

- Introduction to Simulator
- **Environmental Settings**
  - Building SESC executable
  - Running benchmark apps
  - SESC code structure
OS settings

- SESC runs on Linux machines
  - Linux machines with GCC installed

- If you have a SEAS account
  - seas.shell.gwu.edu
  - Redhat Linux 5

- If you don’t have a SEAS account
  - Go and grab one in Tompkins 4th floor, front desk

- If you want to play with SESC on your own machine
  - For Mac
    - install Xcode developer tools package
  - For PC
    - install a virtualization software, e.g., Virtualbox, in your PC
    - In virtualbox, install a Linux virtual machine
Where to download SESC?

- Go to [http://sourceforge.net/projects/sesc/](http://sourceforge.net/projects/sesc/)
  - Click CVS
Download SESC to your Linux Machine

- In your Linux machine, activate a new Linux shell
  - Copy and paste
  - Simply press Enter key
  
  ```bash
cvs -d:pserver:anonymous@sesc.cvs.sourceforge.net:/cvsroot/sesc login
```

- Copy and paste
- Replace `modulename` with `sesc`
  
  ```bash
cvs -z3 -d:pserver:anonymous@sesc.cvs.sourceforge.net:/cvsroot/sesc co -P modulename
```

![Image of terminal output]
Outline

- Introduction to Simulator
- Environmental Settings
- Building SESC executable
- Running benchmark apps
- SESC code structure
Building SESC executable

- **Create new directory in sesc root directory**
  - Give it any name you like, e.g., *run*

- **Build a SESC executable**
  - In the *run* directory, type `../configure` and press ENTER key
  - You find 4 files have been created
    - `Make.defs`  `Makefile`  `config.log`  `config.status`
  - Type `make`
    - GCC will build the `sesc` executable: `sesc.mem`
Troubleshooting

- **For Mac OS users**
  - By default, Mac OS does not come with `cvs` software, you have two options
    - use apple’s own `cvs`, or
    - download the software management tool `fink`, and use fink to install a `cvs` tool
  - When doing the make command, you may run into an error message “… not support x86_64 instructions …”, to solve it
    - edit `/sesc/src/Makefile.defs.Darwin`
    - go to line 55
    - replace “COPTS += -march=pentium-m -mtune=prescott”
    - with “COPTS += -march=core2 -mtune=core2”
Outline

- Introduction to Simulator
- Environmental Settings
- Building SESC executable
- Running benchmark apps
- SESC code structure
Run Benchmark Apps with SESC

- **Go to tests directory**
  - Copy mem.conf and share.conf from confs directory to tests directory

- Three precompiled benchmark Apps
  - crafty, mcf, smatrix

- Running benchmark app with command lines
  - `../run/sesc.mem –cmem.conf crafty > tt.in`
  - `../run/sesc.mem –cmem.conf mcf mcf.in`
  - `../run/sesc.mem –cmem.conf smatrix`

- Every run will generate a report file
  - E.g., sesc_crafty.gdxYmlM [sesc_benchName.randomLetters]
Read Report files

- Get report summary
  - Use the convenient tool `report.pl` to interpret report file
    ```bash
    $ ./scripts/report.pl sesc_crafty.gxYmlM
    ```
  - There are more options in using `report.pl`
    ```bash
    $ ./scripts/report.pl --help
    ```
  - There are a lot of useful info in the report summary
    - Execution time, # of instructions, # of CPU cycles, instruction mix ratios, IPC, different cache miss rates, and etc.

- Get the full report
  - Simply `vim` or `vi` the report file and jump to the entry you want to read
  - E.g. the data cache miss and hit counts
    ```
    P(0)_DL1:writeMiss=31815
    P(0)_DL1:readMiss=57548
    P(0)_DL1:readHit=1989888
    P(0)_DL1:writeHit=1099538
    P(0)_DL1:writeBack=47016
    ```
Outline

- Introduction to Simulator
- Environmental Settings
- Building SESC executable
- Running benchmark apps
- SESC code structure
Source Code Tree

- Modularized source code structure
  - libapp - application interface with SESC
  - libcore – processor core and its components, e.g., branch predictors, reservation stations, pipelines, etc.
  - libemul – MIPS instruction emulation
  - libll – interface between the timing and function simulation parts
  - libmem – non-shared caches
  - libpower – power and energy
  - libsescspot – thermal simulation
  - libsmp – shared memory associated structures (cache coherence)
  - libsuc – profiling classes, and some other special useful classes
Important SESC Classes

- **libcore/Processor.h (Processor.cpp)**
  - Processor::advanceClock()
    - increments the CPU clock of the simulated processor
    - coordinates interactions between different pipeline stages
    - and does the following important work
  - fetch()
    - fetch instructions into the instruction queue
  - issue()
    - issue instructions from the instruction queue into a scheduling window (Reservation Station)
  - retire()
    - retire already executed instructions from the reorder buffer (ROB)
Important SESC Classes

libmem/Cache.h (Cache.cpp)

- Cache::access(MemRequest *mreq)
  - The common interface for accessing caches
  - When called, SESC will figure out the type of the access
    - If read request, call
      - Cache::read(MemRequest *mreq)
    - If write request, call
      - Cache::write(MemRequest *mreq)
    - If a cache writeback request, call
      - Cache::pushLine(MemRequest *mreq)
  - Cache::sendMiss(MemRequest *mreq)
    - This function gets called when cache access turns out to be a miss
    - This is also a virtual function
      - The detailed implementation depends on the type of cache
        - WBCache, WTCache, NICECache (inherited classes)
Other Classes to Look At

- **libmem/mtst1.cpp**
  - The main function entry point

- **libcore/GMemorySystem.cpp**
  - Building all cache-like structures, such as, DL1$, IL1$, L2$, TLBs ...

- **libcore/OSSim.cpp**
  - Acting like an OS, booting and stopping the simulation

- **libcore/RunningProcs.cpp**
  - AdvanceClock () gets called here

- **libcore/MemRequest.cpp**
  - Implements signals that traverse through the memory hierarchy

- **libcore/Gprocessor.cpp**
  - The basic processor components are defined in this class
CallBack Functions

- SESC is an execution-driven simulator
  - Functions are called to simulate parts of the processor every cycle
  - There are other functions called at a later time
    - E.g., the event that missed data is brought back to the cache from the lower level memory

- CallBack class and its subclasses
  - Libsuc/callback.h
    - let the programmer schedule the invocation of a function at a given time in the future
How Does CallBack Work?

- Define the function you want to call in the future
  - E.g., Cache::doRead(MemRequest * mreq) { … }

- Define the callback class that wraps the function
  - E.g., typedef CallbackMember1<Cache, MemRequest *, &Cache::doRead> doReadCB

- Schedule a time to execute the callback function
  - doReadCB::scheduleAbs(nextSlot(), this, mreq)
    - doRead is called at the nextSlot() time
  - Or, doReadCB::schedule(5, this, mreq)
    - doRead is called after 5 clock cycles
Suggestions

- Get some background knowledge on C++ if you need
  - The concept of class, inheritance, virtual function, etc.

- Get familiar with Linux shell commands
  - cd, pwd, ls, grep, etc.

- Read header file first
  - .h file defines the attributes and functions of a class

- Start early