CSE 5095 & ECE 4451 & ECE 5451 – Spring 2017

Assignment 2 (developed by Kamran)

SimpleSim: A Single-Core System Simulator

Marten van Dijk

Syed Kamran Haider, Chenglu Jin, Phuong Ha Nguyen

Department of Electrical & Computer Engineering

University of Connecticut

Based on Pin PLDI Tutorial 2007

by Kim Hazelwood, David Kaeli, Dan Connors, and Vijay Janapa Reddi

Agenda

I. Pin Intro and Overview

II. Fundamentals of Pin based SimpleSim Simulator

III. Assignment 2: Implementing a simple Cache model in SimpleSim

IV. Assignment 3 (Not yet included): Implementing Memory Integrity Verification in SimpleSim

V. Assignment 4 (Not yet included): Demonstrating a Cache Side-Channel Attack in SimpleSim

Part One:
Introduction and Overview

What is Instrumentation?

- A technique that inserts extra code into a program to collect runtime information
- Instrumentation approaches:
  - Source instrumentation:
    - Instrument source programs
  - **Binary instrumentation:** (e.g. using Intel's Pin)
    - Instrument executables directly

Why use Dynamic Instrumentation?

- ✓ No need to recompile or relink
- ✓ Discover code at runtime
- ✓ Handle dynamically-generated code
- ✓ Attach to running processes

How is Instrumentation used in Compiler Research?

Program analysis

- Code coverage
- Call-graph generation
- Memory-leak detection
- Instruction profiling

Thread analysis

- Thread profiling
- Race detection

How is Instrumentation used in Computer Architecture Research?

- Trace Generation
- Branch Predictor and Cache Modeling
- Fault Tolerance Studies
- Emulating Speculation
- Emulating New Instructions

Advantages of Pin Instrumentation

- Easy-to-use Instrumentation:
  - Uses dynamic instrumentation
    - Do not need source code, recompilation, post-linking
- Programmable Instrumentation:
  - Provides rich APIs to write in C/C++ your own instrumentation tools (called Pintools)
- Multiplatform:
  - Supports x86, x86-64, Itanium, Xscale
  - Supports Linux, Windows, MacOS
- Robust:
  - Instruments real-life applications: Database, web browsers, …
  - Instruments multithreaded applications
  - Supports signals
- Efficient:
  - Applies compiler optimizations on instrumentation code

Using Pin

- Launch and instrument an application

```
$ pin -t pintool -- application
```

Instrumentation engine
(provided in the kit)

Instrumentation tool
(write your own, e.g. SimpleSim)

Attach to and instrument an application

```
$ pin -t pintool -pid 1234
```

Pin Instrumentation APIs

- Basic APIs are architecture independent:
  - Provide common functionalities like determining:
    - Control-flow changes
    - Memory accesses
- Architecture-specific APIs
  - e.g., Info about segmentation registers on IA32
- Call-based APIs:
  - Instrumentation routines
  - Analysis routines

Instrumentation vs. Analysis

- Instrumentation routines define where instrumentation is inserted
  - e.g., before instruction
  - ☞ **Occurs *first time* an instruction is executed**
- Analysis routines define what to do when instrumentation is activated
  - e.g., increment counter
  - ☞ **Occurs *every time* an instruction is executed**

Pintool 1: Instruction Count

```
    counter++;
sub $0xff, %edx
    counter++;
cmp %esi, %edx
    counter++;
jle <L1>
    counter++;
mov $0x1, %edi
    counter++;
add $0x10, %eax
```

Application's Instructions

Instructions inserted by Instrumentation tool
(e.g. SimpleSim)

Pintool 1: Instruction Count Output

```
$ /bin/ls
 Makefile imageload.out itrace proccount
 imageload inscount0 atrace itrace.out

$ pin -t inscount0 -- /bin/ls
 Makefile imageload.out itrace proccount
 imageload inscount0 atrace itrace.out
```

- `Count 422838`

ManualExamples/inscount0.cpp

```
#include <iostream>
#include "pin.h"

UINT64 icount = 0;

void docount() { icount++; }
```

analysis routine

```
void Instruction(INS ins, void *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}
```

instrumentation routine

```
void Fini(INT32 code, void *v)
{ std::cerr << "Count " << icount << endl; }

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Pintool 2: Instruction Trace

```
    print_ip(ip);
sub  $0xff, %edx
    print_ip(ip);
cmp  %esi, %edx
    print_ip(ip);
jle  <L1>
    print_ip(ip);
mov  $0x1, %edi
    print_ip(ip);
add  $0x10, %eax
```

Application's Instructions

Instructions inserted by Instrumentation tool (e.g. SimpleSim)

Need to pass IP argument to the analysis routine (i.e. print_ip())

Pintool 2: Instruction Trace Output

```
$ pin -t itrace -- /bin/ls
 Makefile imageload.out itrace proccount
 imageload inscount0 atrace itrace.out
```

- `$ head -4 itrace.out` (printing trace file)

```
0x40001e90
0x40001e91
0x40001ee4
0x40001ee5
```

ManualExamples/inscount0.cpp

Argument to analysis routine

analysis routine

instrumentation routine

```
#include <stdio.h>
#include "pin.H"
FILE * trace;

void printip(void *ip){fprintf(trace,"%p\n",ip);}

void Instruction(INS ins, void *v) {
   INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip,
                                IARG_INST_PTR, IARG_END);
}

void Fini(INT32 code, void *v) { fclose(trace); }

int main(int argc, char * argv[]) {
    trace = fopen("itrace.out", "w");
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);

    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

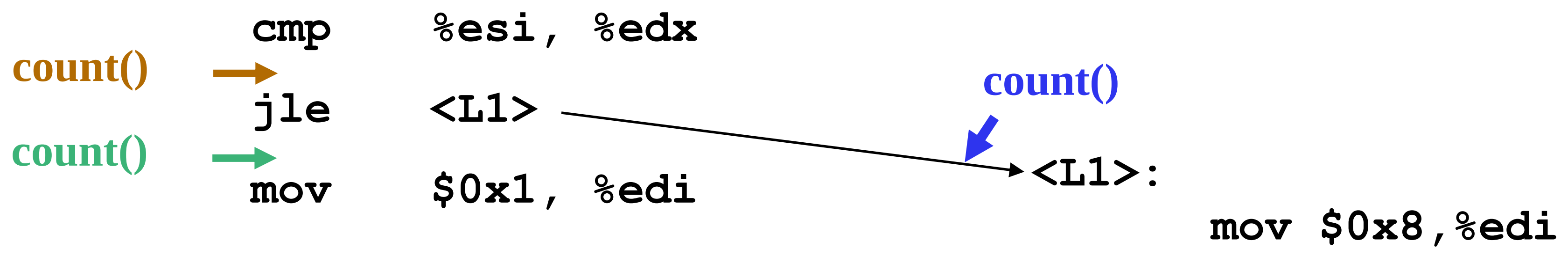
Examples of Arguments to Analysis Routine

- `IARG_INST_PTR`
  - **Instruction pointer (program counter) value**
- `IARG_UINT32 <value>`
  - **An integer value**
- `IARG_REG_VALUE <register name>`
  - **Value of the register specified**
- `IARG_BRANCH_TARGET_ADDR`
  - **Target address of the branch instrumented**
- `IARG_MEMORY_READ_EA`
  - **Effective address of a memory read**

And many more … (refer to the Pin manual for details)

Instrumentation Points

- Instrument points relative to an instruction:
  - *Before (IPOINT_BEFORE)*
  - After:
    a. Fall-through edge (IPOINT_AFTER)
    b. Taken edge (IPOINT_TAKEN_BRANCH)



Part Two:
Fundamentals of SimpleSim Simulator

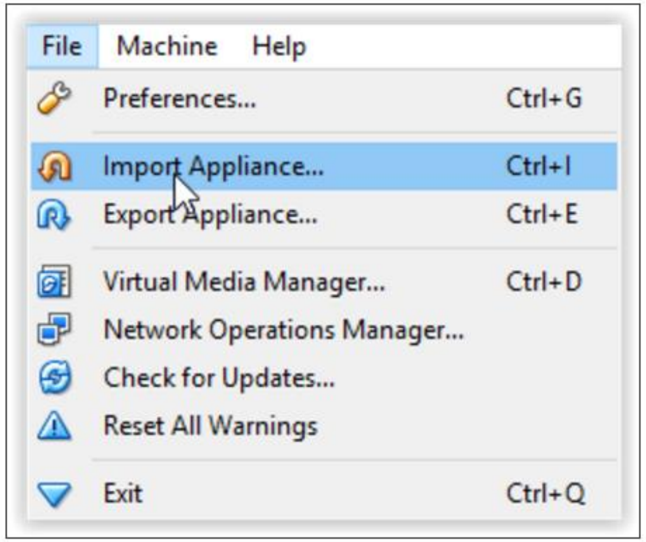
Setting up The System

- Download and Install VirtualBox from here.
- A Ubuntu 14.04 Virtual Machine "ECE5451-VM" is provided.
- Download the ECE5451-VM image from Piazza or the following link: https://drive.google.com/open?id=0B8FDhZrBLHIyMFZXVjVUcEh6Q28
- Import the ECE5451-VM in VirtualBox (see next slides…)

Importing a Virtual Appliance in OVF Format

To Import a Virtual Appliance provided in OVF Format

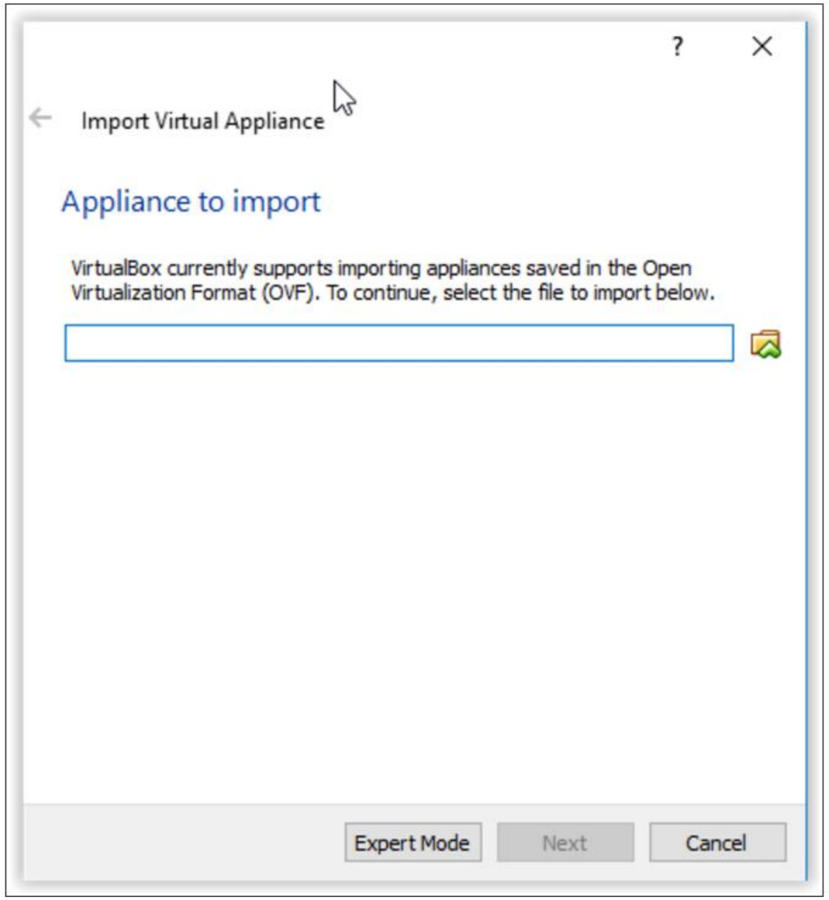
- 1. Select **File | Import Appliance** from the taskbar at the top of the window. The **Appliance Import Wizard** will appear.



Importing a Virtual Appliance in OVF Format

To Import a Virtual Appliance provided in OVF Format

- 2. Open the file dialog and select the **OVF file** with the .ovf file extension.
- 3. The wizard will then display the virtual machines in the OVF file.
  - Users can double-click on the description items to configure the settings of the VM.
  - Upon clicking **Import**, the disk images will be copied and the virtual machines will be created according to the settings that are explained in the dialog.



Setting up SimpleSim Simulator

- Login the ECE5451-VM using the following credentials:
  - Username: student
  - Password: student5451
- You won't need to login as Admin, but in case you do, use the following credentials:
  - Username: Admin
  - Password: Admin5451
- The simulator source code resides in SimpleSim_Public directory.
  - ```
    [student@ECE5451-VM ~]$ cd SimpleSim_Public/
    [student@ECE5451-VM ~/SimpleSim_Public]$
    ```
- Compile the simulator by simply running `make`
  - ```
    [student@ECE5451-VM ~/SimpleSim_Public]$ make
    ```

Running a Test Application

- Currently, a simple application "counters" is added under `tests/benchmarks/counters/counters.cc`
- To run this application, do
  - `[student@ECE5451-VM ~/SimpleSim_Public]$ make counters_bench_test`
- Compile the simulator by simply running `make`. The simulator prints various stats. (For now you'll see Cache Hits and Cache Evictions to be 0 as you need to implement the cache controller…)

```
[student@ECE5451-VM ~/SimpleSim_Public]$ make
Starting Application...
Done...!
*******************************************************
                 SIMULATOR STATISTICS
*******************************************************
           Instructions Count 10894
                Read Accesses 3973
               Write Accesses 1447
                   Cache Hits 3584
                 Cache Misses 389
              Cache Evictions 4
              Total Time (us) 1883644
*******************************************************
make[1]: Leaving directory `/home/student/SimpleSim_Public/tests/benchmarks/counters'
TEST: counters_bench_test PASSED
```

SimpleSim Directory Structure

- **`SimpleSim_Public`** folder contains various subdirectories:
  - **`pin_home`**: Contains Intel's Pin related files. Nothing should be changed here.
  - **`simulator`**: Contains SimpleSim's instrumentation files, e.g. Cache and DRAM models.
    - **`simulator/simulator_main.cc`** is the top level file.
    - **`simulator/Parameters.h`** is the configuration file for the simulator.
  - **`tests`**: Contains application(s) to run using the simulator.

Part Three:
Task 1: Implementing a simple Cache model

Cache Model Requirements

- You need to implement a simple Set-Associative L1 Cache.
  - The system two level memory hierarchy: L1 Cache and DRAM
  - The system is single-core, hence no cache coherence protocol required.
- The Cache Model requires the following features:
  - Configurable cache line size, capacity & associativity.
  - “Least Recently Used” replacement policy.
  - Only ‘dirty’ cache lines evicted from the cache need to be written back to DRAM.
- All configuration parameters can be added to Parameters.h file.

Cache Model Implementation

- `simulator/SimpleCache.h` provides an interface to the simulator via `SimpleCache` class.
  - **`bool Read(uint64_t addr, uint8_t* data_buf);`**
    - Return '*false*' if Cache miss; '*true*' if Cache Hit.
    - The data read from cache should be copied into data_buf.
  - **`uint64_t Write(uint64_t addr, uint8_t* data_buf);`**
    - The data to be written in cache is provided in data_buf.
    - If cache line 'addr' already exists in cache, update it and set dirty bit. No eviction, so return NULL.
    - If cache line 'addr' is not present in cache, insert it. Return evicted address if an eviction of a **dirty cache line** happens. Evicted data should be stored in data_buf. Return NULL if no eviction happens.
  - `data_buf` size is equal to CACHELINE_SIZE configured in Parameters.h
- Implement these functions in `simulator/SimpleCache.cpp` file.
- A correct implementation should result in about the same number of cache hits/miss/evictions for the same application every time.