6.823: Computer System Architecture

Introduction to PIN

Hyun Ryong (Ryan) Lee
6823-tas@csail.mit.edu

Adapted from: Prior 6.823 offerings, and Intel's Tutorial at CGO 2010

Designing Computer Architectures

Build computers that run programs *efficiently*

Two important components:

1. Study of programs and patterns
   – Guides interface, design choices
2. Engineering under constraints
   – Evaluate tradeoffs of architectural choices

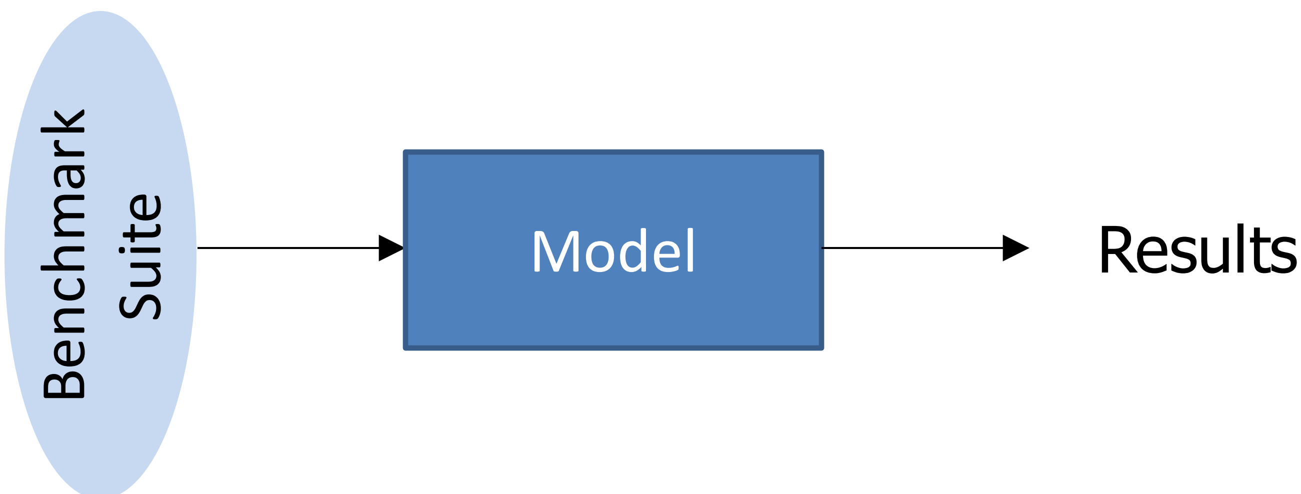
Simulation: An Essential Tool in Architecture Research

- A tool to reproduce the behavior of a computing device

- Why use simulators?
  - Obtain fine-grained details about internal behavior
  - Enable software development
  - Obtain performance predictions for candidate architectures
  - Cheaper than building system

Labs

- Focus on understanding program behavior, evaluating architectural tradeoffs



PIN

- www.pintool.org
  - Developed by Intel
  - Free for download and use

- A tool for dynamic binary instrumentation

Runtime

No need to re-compile or re-link

Insert code in the program to collect information

Pin: A Versatile Tool

- Architecture research
  - Simulators: zsim, CMPsim, Graphite, Sniper, Swarm
- Software Development
  - Intel Parallel Studio, Intel SDE
  - Memory debugging, correctness/perf. analysis
- Security
  - Taint analysis

Useful tool to have in your arsenal!

PIN

- www.pintool.org
  - Developed by Intel
  - Free for download and use

- A tool for dynamic binary instrumentation

Runtime

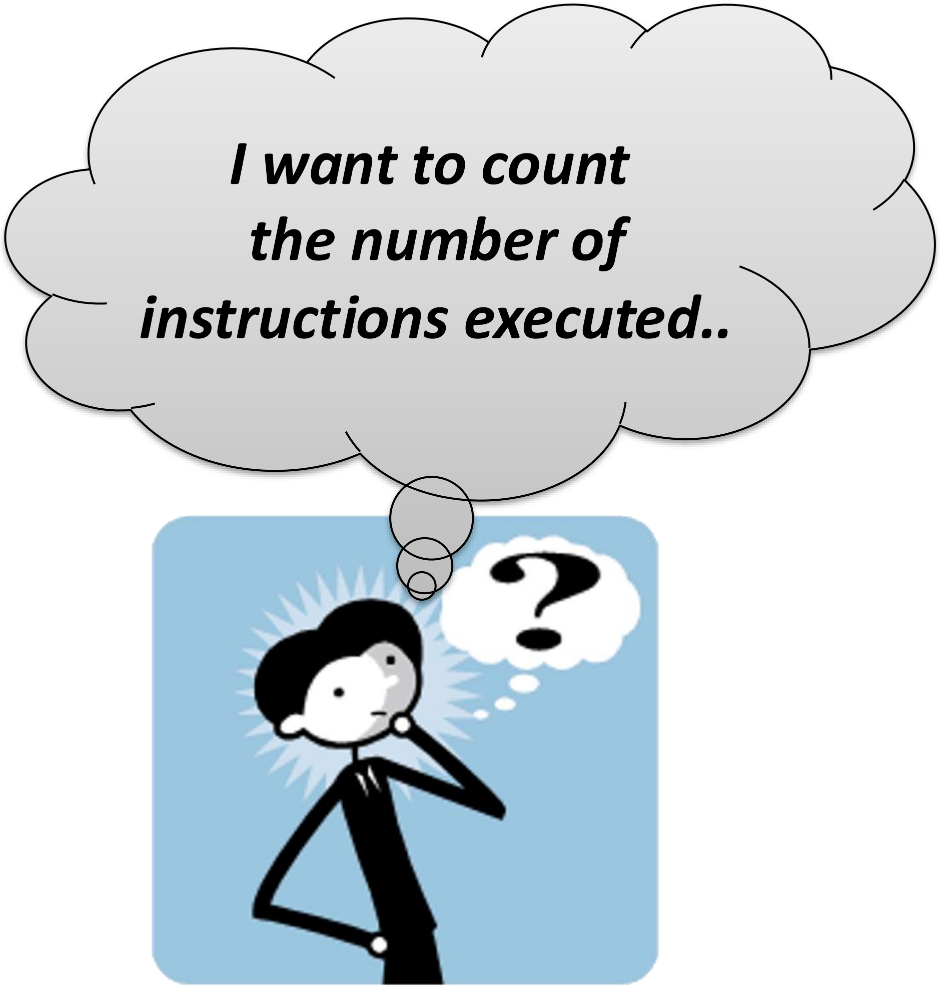
No need to re-compile or re-link

Insert code in the program to collect information

Instrumenting Instructions

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

Example 1: Instruction Count

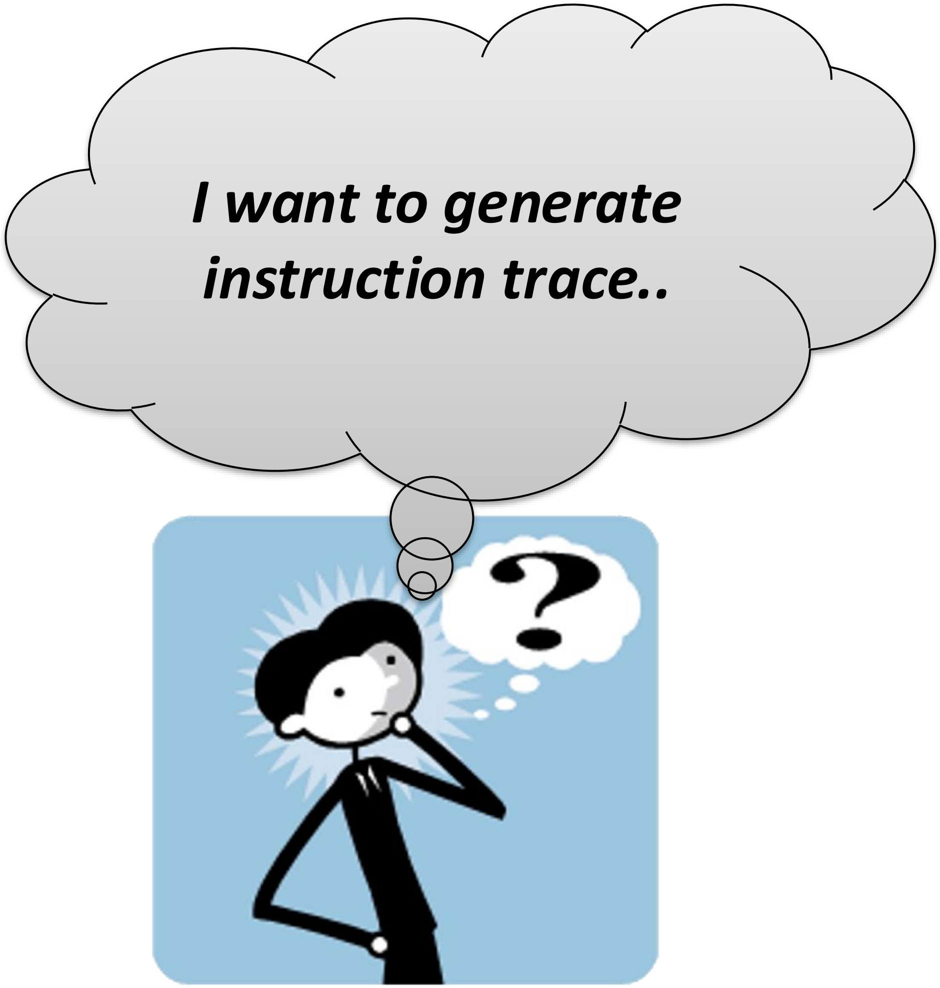



```
sub  $0xff, %edx
cmp  %esi, %edx
jle  <L1>
mov  $0x1, %edi
add  $0x10, %eax
```

Example 1: Instruction Count


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

Example 2: Instruction Trace




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

Example 2: Instruction Trace

Let's print instruction pointers before every instruction!

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

Example 2: Instruction Trace

Let’s print
instruction pointers
before every instruction!

```
Print(ip);
sub $0xff, %edx
```

This is “Instrumentation”

```
jle <L1>
Print(ip);
mov $0x1, %edi
Print(ip);
add $0x10, %eax
```

What is Instrumentation?

- A technique that inserts extra code into a program to collect runtime information
  - Program Analysis: performance profiling, error detection, capture and replay
  - Architectural study: processor and cache simulation, trace collection
- Instrumentation approaches:
  - Source instrumentation:
    - Instrument source programs
  - **Binary instrumentation**:
    - Instrument executables directly

What can you do with Pin?

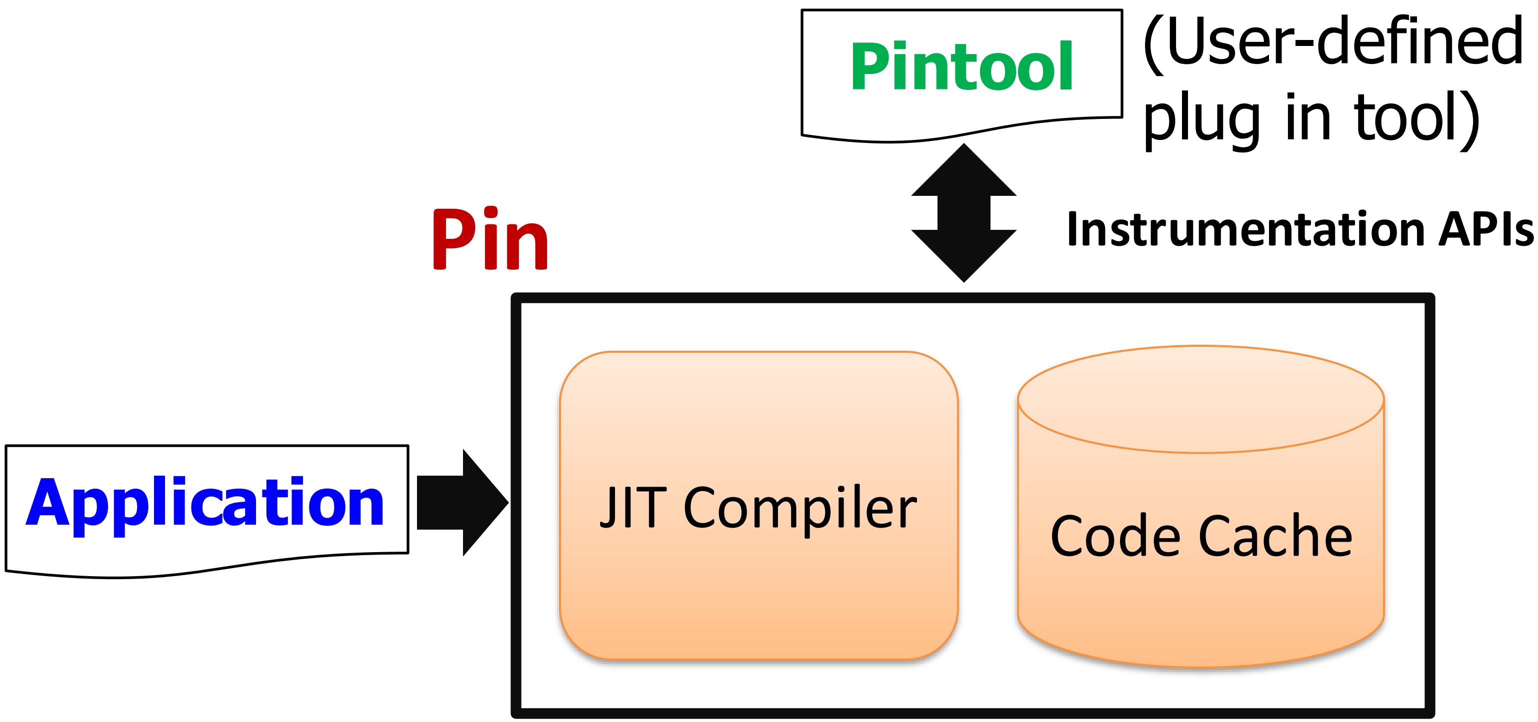
- Pin gives you the ability to
  - inspect every instruction, and then
    - insert extra code (optional)
    - executed the inserted code

Instrumentation routine

Analysis routine

How to use Pin

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



Advantages of Pin

- Dynamic Instrumentation
  – No need for source code, re-compilation or post-linking
- Programmable Instrumentation
  – Provides rich APIs to write in C/C++ your own instrumentation tools (called Pintools)
- Multiplatform
  – Supports IA-32, IA-64, Itanium
  – Supports Linux, Windows, MacOS
- Robust
  – Instrument real life applications: web browsers, databases
  – Instrument multithreaded applications
- If you can run it, you can Pin it

How is Instrumentation used in Computer Architecture?

- Trace Generation
- Branch Predictor and Cache Modeling
- Fault Tolerance Study
- Emulating Speculation
- Emulating New Instructions
- Cache Coherence Protocols

Writing Pintools

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

Example 1: Instruction Count

Let’s increment counter by one before every instruction!

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

Example 1: Instruction Count

Let's increment counter by one ***before every instruction!***

Instrumentation routine

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

Example 1: Instruction Count

Analysis routine

Instrumentation routine

Let's increment counter by one before every instruction!

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

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
```

Pintool 1: Instruction Count Output

```
$ /bin/ls

  Makefile atrace.o imageload.out itrace
  proccount Makefile.example imageload
  inscount0 itrace.o proccount.o atrace
  imageload.o inscount0.o itrace.out


$ pin –ifeellucky -t inscount0 -- /bin/ls

  Makefile atrace.o imageload.out itrace
  proccount Makefile.example imageload
  inscount0 itrace.o proccount.o atrace
  imageload.o inscount0.o itrace.out

Count 422838
```

ManualExamples/inscount0.C

```
#include <iostream>
#include "pin.h"
UINT64 icount = 0;
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, “pintool”, “o”,
                           “results.out”, “specify output file”);
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)
{ FILE* outfile = fopen(KnobOutputFile.Value().c_str(),”w”);
  fprintf(outfile, “Count %d\n”, icount);}

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

ManualExamples/inscount0.C

```
#include <iostream>
#include "pin.h"
UINT64 icount = 0;
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, “pintool”, “o”,
                            “results.out”, “specify output file”);
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)
{ FILE* outfile = fopen(KnobOutputFile.Value().c_str(),”w”);
  fprintf(outfile, “Count %d\n”, icount);}

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

Instrumentation Points

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

```
          cmp   %esi, %edx      count()
count() →
          jle   <L1>  ────────────→  <L1>:
count() →
          mov   $0x1, %edi              mov $0x8,%edi
```

Pintool 2: Instruction Trace

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

Pintool 2: Instruction Trace Output

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

$ head -4 itrace.out
0x40001e90
0x40001e91
0x40001ee4
0x40001ee5
```

ManualExamples/itrace.C

argument to analysis routine

```
#include <stdio.h>
#include "pin.H"
FILE * trace;
void printip(void *ip) { fprintf(trace, "%p\n", ip); }
```

analysis routine

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

instrumentation routine

```
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_PTR <pointer>`**
  - A pointer to some data
- **`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)

Modifying Program Behavior

- Pin allows you not only to *observe*, but also *change* program behavior
- Ways to change program behavior:
  - Add/delete instructions
  - Change register values
  - Change memory values
  - Change control flow
  - Inject errors

Writing **Efficient** Pintools

(we will cover this in detail next week)

Reducing Instrumentation Overhead

Total Overhead = Pin's Overhead + Pintool's Overhead

- The job of Pin developers to minimize this
- ~5% for SPECfp and ~20% for SPECint

Reducing Instrumentation Overhead

Total Overhead = Pin's Overhead + Pintool's Overhead

- The job of Pin developers to minimize this
- ~5% for SPECfp and ~20% for SPECint

- Pintool writers can help minimize this!

Reducing Pintool's Overhead

Pintool's Overhead

Instrumentation Routines Overhead + Analysis Routines Overhead

Frequency of calling an Analysis Routine x Work required in the Analysis Routine

Next week, we will see
how we can reduce these overheads

Conclusions

- **Instrumentation** is a technique for inserting extra code into a program to observe its behavior
- **Pin** is a dynamic binary instrumentation system
- Instrumentation tools (**Pintools**) are written in C/C++ using Pin's rich API
- **Instrumentation routines** define where instrumentation is **inserted**
- **Analysis routines** define what to do when instrumentation is **activated**

Conclusions

- **Instrumentation** is a technique for inserting extra code into a program to observe its behavior
- **Pin** is a dynamic binary instrumentation system
- Instrumentation tools (**Pintools**) are written in C/C++

Lab 0 released on website

- Instrumentation routines define where instrumentation is **inserted**
- **Analysis routines** define what to do when instrumentation is **activated**