CIS 429/529

Pin Introduction

Prof. Michel A. Kinsky
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:
    - Instrument executables directly
Example: Instruction Count

Logically

Add counter, 0x1

Actually

counter ++
sub $0xff, %edx

counter ++
cmp %esi, %edx

counter ++
jle <L1>
counter ++
mov $0x1, %edi
counter ++
add $0x10, %eax
How Pin Works – High Level

• What is modified
  ‣ New instructions are added at user defined points
  ‣ Static addresses and references
  ‣ Register allocation
  ‣ Pin stack

• What is executed
  ‣ Instrumented traces
  ‣ Code cache
How Pin Works – High Level

• When does the modification occur
  ‣ At run time
  ‣ Can attach to running process
Example: 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
```
Instrumentation vs. Simulation

• Advantages of Simulation:
  ‣ Detailed modeling of processors
  ‣ Can model non-existing hardware

• Advantages of Instrumentation:
  ‣ Easy to prototype
  ‣ Fast to run (allowing complete runs)
Usage in Architecture

• 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
What is Pin?

• 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 IA-32, EM64T, Itanium, Xscale
  ‣ Supports Linux, Windows, MacOS
What is Pin?

• Robust:
  ‣ Instruments real-life applications
    • Database, search engines, web browsers, …
  ‣ Instruments multithreaded applications

• Efficient:
  ‣ Applies compiler optimizations on instrumentation code
How to use Pin?

- Launch and instrument an application
  
  \$ pin -t pintool - application

- 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

- Concepts borrowed from the ATOM tool:
  - **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
$ /bin/ls

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

$ pin -t inscount0 -- /bin/ls

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

Count 422838
#include <iostream>
#include "pin.h"

UINT64 icount = 0;
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "results.out", "specify output file");

void docount() { icount++;

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

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;
}
• Same source code works on the 4 architectures

• Pin automatically and efficiently saves/restores application state
Pintool 2: Instruction Trace

- Need to pass an argument (ip) to the analysis routine (printip())

```
Print(ip)
sub $0xff, %edx
Print(ip)
cmp %esi, %edx
Print(ip)
jle <L1>
Print(ip)
mov $0x1, %edi
Print(ip)
add $0x10, %eax
```
$ pin -t itrace -- /bin/ls

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

$ head -4 itrace.out

0x40001e90
0x40001e91
0x40001ee4
0x40001ee5
#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;
}
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
Arguments to Analysis Routine

• 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:
    - Fall-through edge (IPOINT_AFTER)
    - Taken edge (IPOINT_TAKEN)

```
cmp %esi, %edx
jle <L1>

mov $0x1, %edi

<count()>
```

```
<count()>
<count()>
vect

mov $0x8, %edi
```
Instrumentation Granularity

• Instrumentation with Pin can be done at 3 different granularities:
  ▸ Instruction
  ▸ Basic block
    • A sequence of instructions terminated at a (conditional or unconditional) control-flow changing instruction
    • Single entry, single exit
  ▸ Trace
    • A sequence of basic blocks terminated at an unconditional control-flow changing instruction
    • Single entry, multiple exits
Instrumentation Granularity

- 1 Trace, 2 basic blocks, 6 instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>$0xff, %edx</td>
</tr>
<tr>
<td>Cmp</td>
<td>%esi, %edx</td>
</tr>
<tr>
<td>jle</td>
<td>&lt;L1&gt;</td>
</tr>
<tr>
<td>mov$</td>
<td>0x1, %edi</td>
</tr>
<tr>
<td>add</td>
<td>$0x10, %eax</td>
</tr>
<tr>
<td>jmp</td>
<td>&lt;L2&gt;</td>
</tr>
</tbody>
</table>
Recap of Pintool 1: Instruction Count

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

Straightforward, but the counting can be more efficient
Faster Instruction Count

- Reduce the number of calls made to analysis routine

```
counter += 3
sub $0xff, %edx

cmp %esi, %edx

jle <L1>

mov $0x1, %edi

add $0x10, %eax
```
#include <iostream>
#include "pin.h"

UINT64 icount = 0;

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o",
    "results.out", "specify output file");

void docount(INT32 c) { icount += c; }  

void Trace(TRACE trace, void *v) {
    for (BBL bbl = TRACE_BblHead(trace);
        BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
        BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)docount,
                        IARG_UINT32, BBL_NumIns(bbl), IARG_END);
    }
}

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;
}
Modifying Program Behavior

• Pin allows you not only to observe but also to change program behavior

• Ways to change program behavior:
  ‣ Add/delete instructions
  ‣ Change register values
  ‣ Change memory values
  ‣ Change control flow
  ‣ Inject errors
Example: Emulation of Loads

```
sub    $0x11c,%esp

mov    0xc(%ebp),%eax

add    $0x128, %eax

mov    0x8(%ebp),%edi

xor    %eax, %edi
```
Multithreading Support

- Notify the pintool when a thread is created or exited
- Provide a “thread id” for pintools to identify a thread
- Provide locks for pintools to access shared data structures
$ pin -mt -t mtest -- thread
Creating thread
Creating thread
Joined 0
Joined 1

$ cat mtest.out
0x400109a8: 0
thread begin 1 sp 0x80acc00 flags f00
0x40001d38: 1
thread begin 3 sp 0x43305bd8 flags f21
0x40011220: 3
thread begin 2 sp 0x42302bd8 flags f21
0x40010e15: 2
0x40005cdc: 2
thread end 3 code 0
0x40005e90: 0
0x40005e90: 0
thread end 2 code 0
thread end 1 code 0
Debugging Pintools

- Invoke gdb with your pintool (but don’t use “run”)

```sh
$ gdb inscount0
(gdb)
```

- On another window, start your pintool with “-pause_tool”

```sh
$ pin -pause_tool 5 -t inscount0 -- /bin/ls
Pausing to attach to pid 32017
```

- Go back to gdb:
  - Attach to the process
  - Use “cont” to continue execution; can set breakpoints as usual

```sh
(gdb) attach 32017
(gdb) break main
(gdb) cont
```
Conclusions

• Pin
  ‣ Build your own architectural tools with ease
  ‣ Run on multiple platforms:
    • IA-32, EM64T, Itanium, and XScale
    • Linux, Windows, MacOS
  ‣ Work on real-life applications
  ‣ Efficient instrumentation
Reducing Pintool’s Overhead

Pintool’s Overhead

Instrumentation Routines Overhead + Analysis Routines Overhead

- Frequency of calling an Analysis Routine \times \text{Work required in the Analysis Routine}

- \text{Work required for transiting to Analysis Routine} + \text{Work done inside Analysis Routine}
• Reducing Frequency of Calling Analysis Routines

  ‣ Key:

    • Instrument at the largest granularity whenever possible:

      ‣ Trace > Basic Block > Instruction