Introduction to the Pin Instrumentation Tool

Quan Jia

Mar 27, 2013

What is Pin?

Pin is Intel's dynamic binary instrumentation engine.

What is Instrumentation?

- A technique that inserts extra code into a program to collect runtime information.
  - Program analysis : performance profiling, error detection, capture & replay
  - Architectural study : processor and cache simulation, trace collection
  - Binary translation : Modify program behavior, emulate unsupported instructions

Instrumentation Approaches

- Source Code Instrumentation (SCI)
  - instrument source programs

- Binary Instrumentation (BI)
  - instrument binary executable directly

SCI Example (Code Coverage)

Original Program

```
void foo() {
  bool found=false;
  for (int i=0; i<100; ++i) {
     if (i==50) break;
     if (i==20) found=true;
  }
  printf("foo\n");
}
```

Instrumented Program

```
char inst[5];
void foo() {
  bool found=false; inst[0]=1;
  for (int i=0; i<100; ++i) {
     if (i==50) { inst[1]=1;break;}
     if (i==20) { inst[2]=1;found=true;}
     inst[3]=1;
  }
printf("foo\n");
inst[4]=1;
}
```

Binary Instrumentation (BI)

- Static binary instrumentation – inserts additional code and data <u>before execution</u> and generates a persistent modified executable

- Dynamic binary instrumentation – inserts additional code and data <u>during execution</u> without making any permanent modifications to the executable.

BI Example – Instruction Count

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

Where Innovation Is Tradition

BI 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
```

Advantages

- Binary instrumentation
  - Language independent
  - Machine-level view
  - Instrument legacy/proprietary software
- Dynamic instrumentation
  - No need to recompile or relink
  - Discover code at runtime
  - Handle dynamically-generated code
  - Attach to running processes

What is Pin?

Pin is Intel's dynamic binary instrumentation engine.

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
  - OS's: Windows, Linux, OSX, Android
- Robust:
  - Instruments real-life applications: Database, web browsers, …
  - Instruments multithreaded applications
  - Supports signals
- Efficient:
  - Applies compiler optimizations on instrumentation code

Widely Used and Supported

- Large user base in academia and industry
  - 30,000+ downloads
  - 700+ citations
  - Active mailing list (Pinheads)
- Actively developed at Intel
  - Intel products and internal tools depend on it
  - Nightly testing of 25000 binaries on 15 platforms

Using Pin

Launch and instrument an application

```
$ pin -t pintool.so -- application
```

Instrumentation engine

(provided in the kit)

Instrumentation tool

(write your own, or use one provided in the kit)

Attach to and instrument an application

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

Pin and Pintools

- Pin – the instrumentation engine
- Pintool – the instrumentation program

- Pin provides the framework and API, Pintools run on Pin to perform meaningful tasks.

- Pintools
  - Written in C/C++ using Pin APIs
  - Many open source examples provided with the Pin kit
  - Certain Do's and Don'ts apply

Pin Instrumentation Capabilities

- Replace application functions with your own.
- Fully examine any application instruction – insert a call to your instrumenting function whenever that instruction executes.
- Pass a large set of supported parameters to your instrumenting function.
  - Register values (including IP), Register values by reference (for modification)
  - Memory addresses read/written by the instruction
  - Full register context
- Track function calls including syscalls and examine/change arguments.
- Track application threads.
- Intercept signals.
- Instrument a process tree.

………

Hands-on Task

- Download the latest Pin from http://www.pintool.org
  - For Windows: make sure you download the correct version that matches your Visual Studio IDE.
- Build all included Pintools under

  source/tools/SimpleExamples
  source/tools/ManualExmaples
- Refer to the user's manual for detailed instructions
  - Attention: Nmake does not work for Windows, use Cygwin to install GNU make instead.

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: Invocation

- Windows examples:

```
> pin.exe -t inscount0.dll -- dir.exe
> pin.exe -t inscount0.dll -o incount.out -- gzip.exe FILE
```

- Linux examples:

```
$ pin -t inscount0.so -- /bin/ls
$ pin -t inscount0.so -o incount.out -- gzip FILE
```

Pintool 1: 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

```
switch to pin stack
save registers
call docount
restore registers
switch to app stack
```

```
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;
}
```

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

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: ManualExamples/itrace.cpp

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_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:
  - Fall-through edge (IPOINT_AFTER)
  - **Taken edge (IPOINT_TAKEN)**

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

Instrumentation Granularity

Instrumentation can be done at three different granularities:

- **Instruction**
- **Basic block**
  - A sequence of instructions terminated at a 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

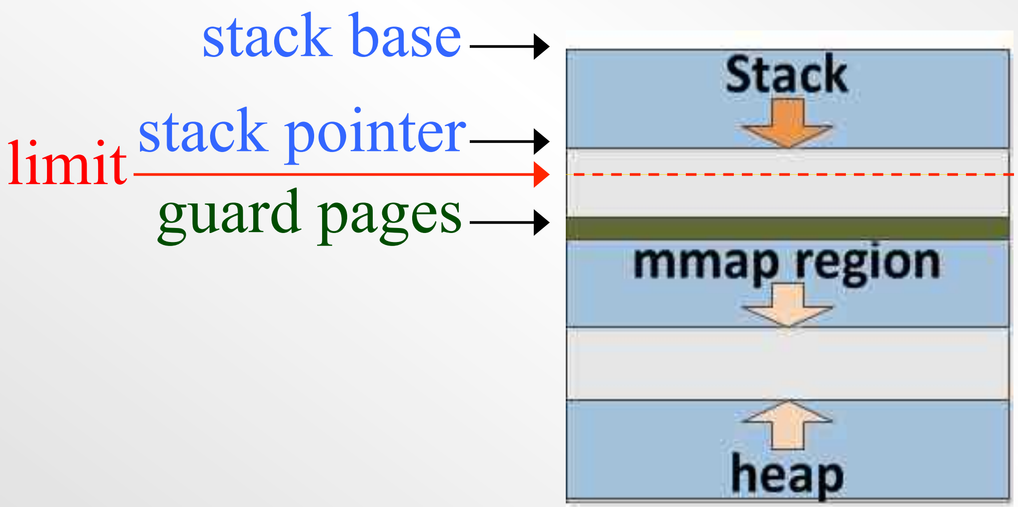
```
sub    $0xff, %edx
cmp    %esi, %edx
jle    <L1>
```

```
mov    $0x1, %edi
add    $0x10, %eax
jmp    <L2>
```

1 Trace, 2 BBs, 6 insts

Hands-on Task: Stack Monitor

- **Goal:** Monitor runtime stack usage and alert if it exceeds a pre-defined limit.
- Process address space:



Hands-on Task: Stack Monitor

- **Steps:**

1. Obtain stack base address when process starts.
2. Perform instruction-level instrumentation.
3. Get runtime stack size (stack_base – stack_pointer).
4. Compare stack size with supplied size limit.

- **Hint:** refer to ManualExamples/stack-debugger.cpp