Dynamic Binary Instrumentation: Introduction to Pin

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

```
Max = 0;
for (p = head; p; p = p->next)
{

   if (p->value > max)
   {

     max = p->value;
   }
}
```

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

```
Max = 0;
for (p = head; p; p = p->next)
{
   printf("In loop\n");
   if (p->value > max)
   {
      printf("True branch\n");
      max = p->value;
   }
}
```

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

```
Max = 0;
for (p = head; p; p = p->next)
{
   count[0]++;
   if (p->value > max)
   {
      count[1]++;
      max = p->value;
   }
}
```

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

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

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

- It executes as a part of the normal instruction stream
- It doesn't modify the semantics of the program

When is instrumentation useful?

- Profiling for compiler optimization/performance profiling:
  - Instruction profiling
  - Basic block count
  - Value profile
- Bug detection/Vulnerability identification/Exploit generation:
  - Find references to uninitialized, unallocated addresses
  - Inspect arguments at a particular function call
  - Inspect function pointers and return addresses
  - Record & replay
- Architectural research: processor and cache simulation, trace collection

Instrumentation

- **Static instrumentation** – instrument before runtime
  - Source code instrumentation
    - Instrument source programs (e.g., clang's source-to-source transformation)
  - IR instrumentation
    - Instrument compiler-generated IR (e.g., LLVM)
  - Binary instrumentation
    - Instrument executables directly by inserting additional assembly instructions (e.g., Dyninst)

- **Dynamic binary instrumentation** – instrument at runtime
  - Instrument code just before it runs (Just in time – JIT)
  - E.g., Pin, Valgrind, DynamoRIO, QEMU

Why **binary** instrumentation

- Libraries are a big pain for source/IR-level instrumentation
  – Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)
- Easily handles multi-lingual programs
  – Source code level instrumentation is heavily language dependent.
- Worms and viruses are rarely provided with source code
- Turning off compiler optimizations can maintain an almost perfect mapping from instructions to source code lines

Dynamic binary instrumentation

- **Pros**
  - No need to recompile or relink
  - Discovers code at runtime
  - Handles dynamically generated code
  - Attaches to running processes (some tools)
- **Cons**
  - Usually higher performance overhead
  - Requires a framework which can be detected by malware

Pin

A Dynamic Binary Instrumentation Tool

1. What can we do with Pin?
2. How does it work?
3. Examples (original Pin examples)
4. Performance overhead
5. Debugging pintools

Pin

- Pin is a tool for the instrumentation of programs. It supports Linux* and Windows* executables for x86, x86_64, and IA-64 architectures.
- Pin allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.

What can we do with Pin?

- Fully examine any (type of) x86 instruction
  - Insert a call to your own function which gets called when that instruction executes
    - Parameters: register values (including IP), memory addresses, memory contents…
- Track function calls, including library calls and syscalls
  - Examine/change arguments
  - Insert function hooks: replace application/library functions with your own
- Track application threads
- And more ☺

If Pin doesn't have it, you don't want it ;)

Advantages of Pin

- **Easy-to-use Instrumentation:**
  - Uses dynamic instrumentation
    - Does 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
  - Supports Linux, Windows binaries
- **Robust**:
  - Instruments real-life applications: Database, web browsers,. . .
  - Instruments multithreaded applications
  - Supports signals
- **Efficient**:
  - Applies compiler optimizations on instrumentation code

Usage of Pin at Intel

- Profiling and analysis products
  - Intel Parallel Studio
    - Amplifier (Performance Analysis)
      - Lock and waits analysis
      - Concurrency analysis
    - Inspector (Correctness Analysis)
      - Threading error detection (data race and deadlock)
      - Memory error detection

GUI
Algorithm
PinTool
Pin

- Architectural research and enabling
  - Emulating new instructions (Intel SDE)
  - Trace generation
  - Branch prediction and cache modeling

Pin usage outside Intel

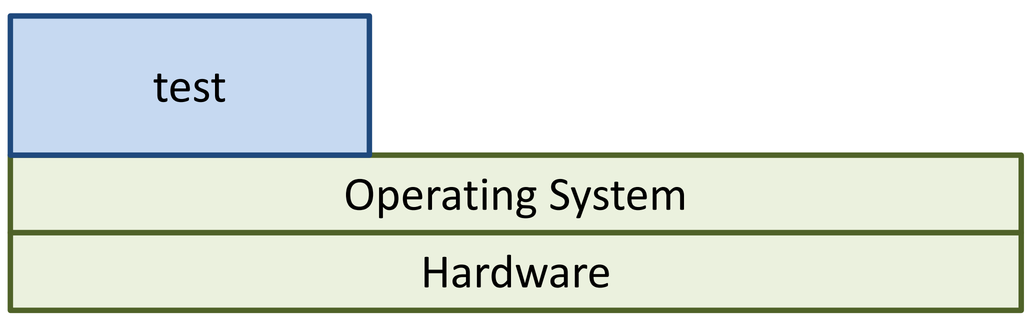
- **Popular and well supported**
  - 100,000+ downloads,
  - 3,500+ citations
  - (as of 2018)

- **Free DownLoad**
  - **www.pintool.org**
  - **Includes: Detailed user manual, source code for 100s of Pin tools**

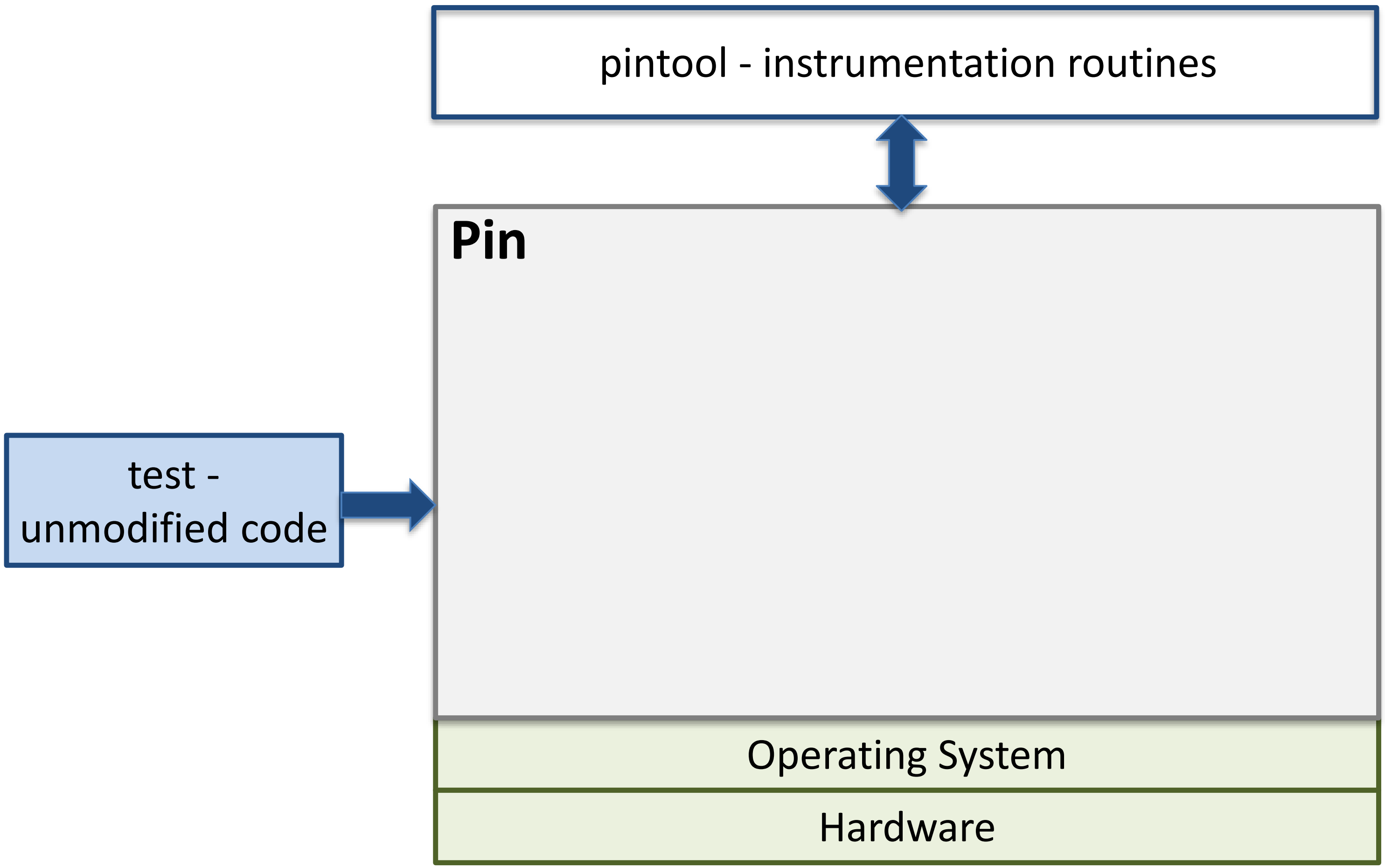
- **Pin User Group (PinHeads)**
  - **http://tech.groups.yahoo.com/group/pinheads/**
  - **Pin users and Pin developers answer questions**

Architecture overview

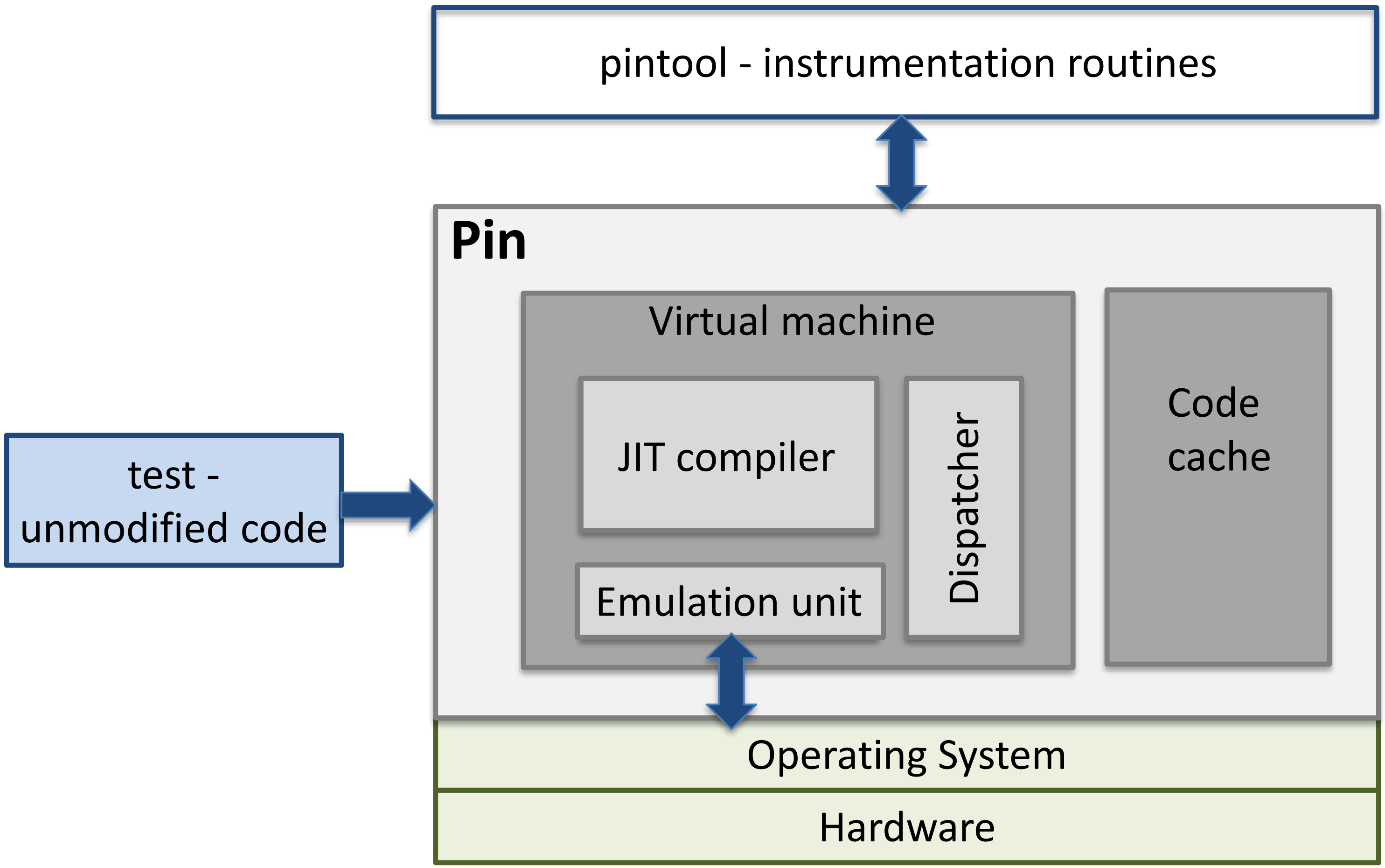
./test



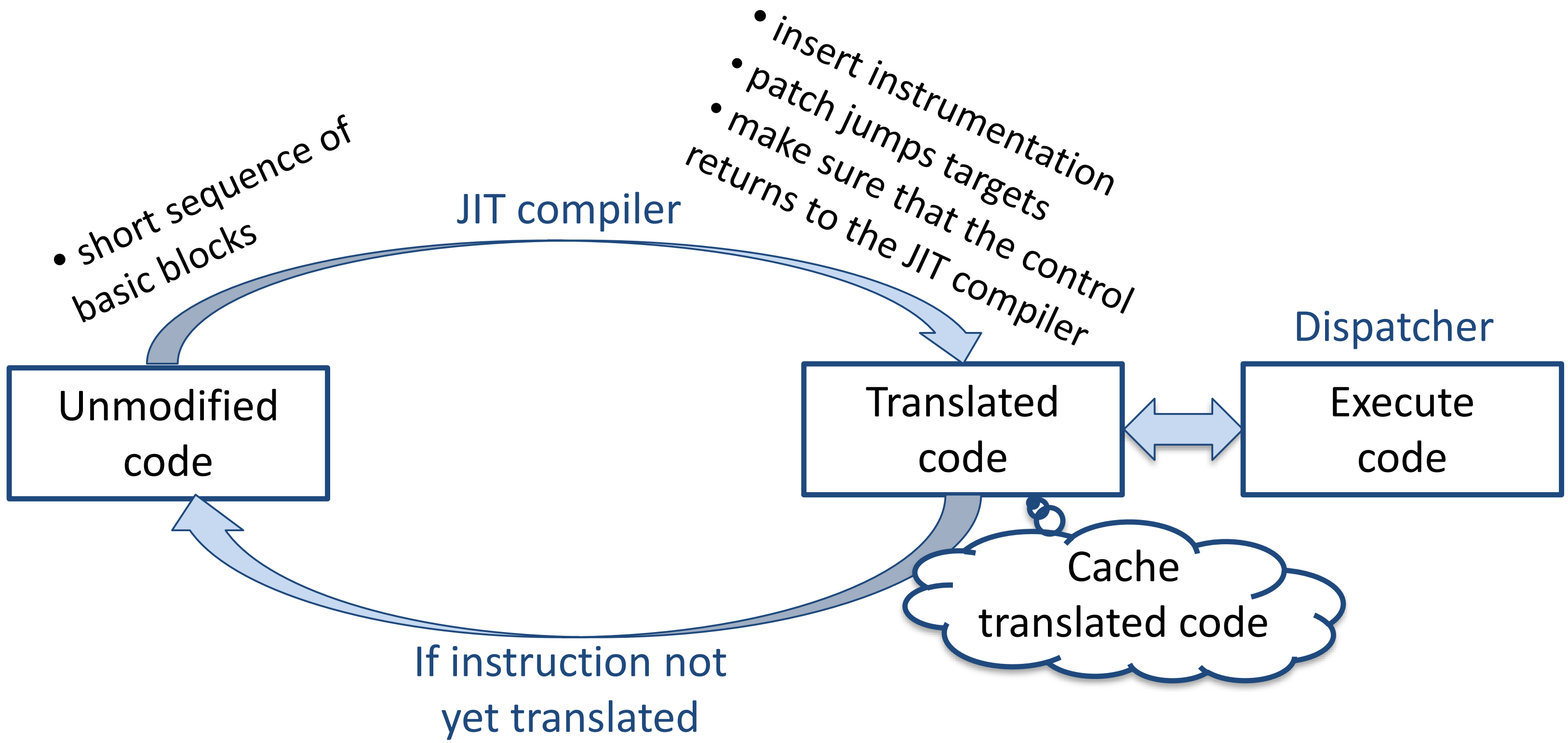
./pin -t pintool -- test



./pin -t pintool -- test



JIT compilation



Example 1: docount
- instruction counting tool

Instruction counting tool

```
#include “pin.h”
uint64_t icount = 0;

void docount() { icount++; }

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

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(); // never returns
   return 0;
}
```

Instruction counting tool

```
#include “pin.h”
uint64_t icount = 0;

void docount() { icount++; }

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

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(); // never returns
   return 0;
}
```

Initialize PIN

Instruction counting tool

```
#include "pin.h"
uint64_t icount = 0;

void docount() { icount++; }

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

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(); // never returns
   return 0;
}
```

`INS` is valid only inside this routine.

Instrumentation routine; called during jitting of INS.

Register instruction instrumentation routine

Instruction counting tool

```
#include "pin.h"
uint64_t icount = 0;

void docount() { icount++; }

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

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(); // never returns
   return 0;
}
```

Analysis routine; Executes each time jitted INStruction executes.

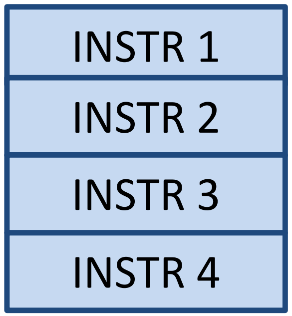
Instruction counting tool

```
#include “pin.h”
uint64_t icount = 0;

void docount() { icount++; }

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

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(); // never returns
   return 0;
}
```

Question: which function gets executed more often?

Instruction counting tool

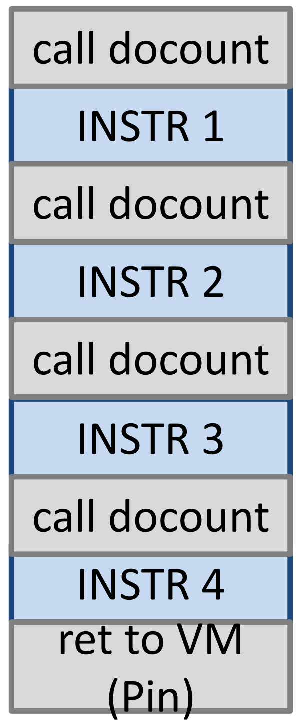
Native BB



Analysis routine (`docount`)

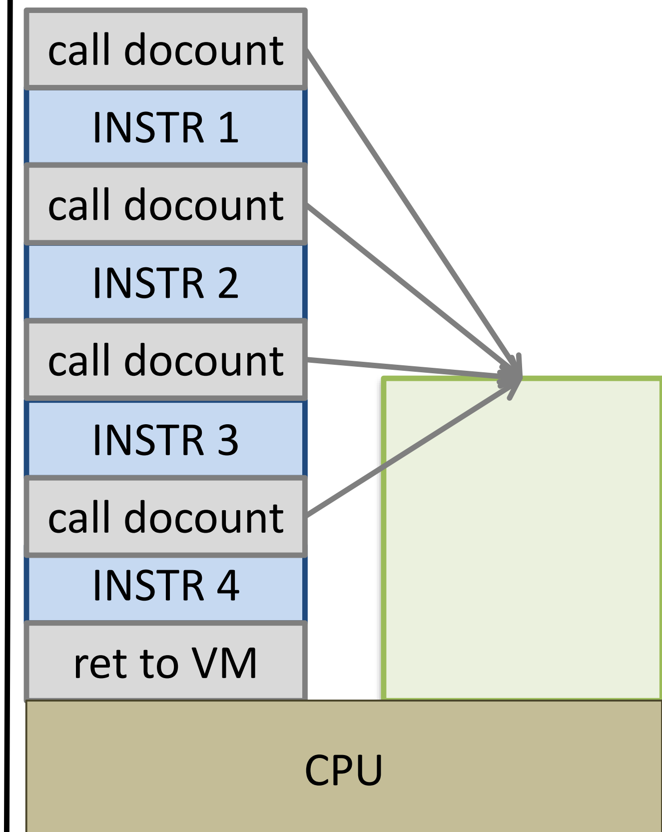
Translation time (JIT compilation)

Rewriting only! We don't execute the native BB now!



Call the `Instruction` **instrumentation routine** to see that we need to **insert a call** to `docount` before every instruction.
To be precise, `INSTR X` are not necessarily exactly the same, but very little is changed.

Execution time (Dispatcher)



Execute the translated block. During the execution, the **analysis routines are executed**. In our case - `docount`.

Instrumentation vs Analysis

- **Instrumentation routines**
  - Define where instrumentation is inserted, e.g., before instruction
  - Invoked when **an instruction is being jitted**
- **Analysis routines**
  - Define what to do when instrumentation is activated, e.g., increment counter
  - Invoked every time **an instruction is executed**

Instruction counting tool

```
#include "pin.h"
uint64_t icount = 0;

void docount() { icount++; }

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

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(); // never returns
   return 0;
}
```

```
  switch to pin stack
  save registers
  call docount
  restore registers
  switch to app stack
• sub   $0xff, %edx
  inc icount
• cmp   %esi,  %edx
  save eflags
  inc icount
  restore eflags
• jle   <L1>
  inc icount
• mov   0x1, %edi
```

Pin execution

Pin execution

1. Download Pin from http://www.pintool.org

```
asia@makai:~/vu/pin-2.11-49306-gcc.3.4.6-ia32_intel64-linux$ pwd
/home/asia/vu/pin-2.11-49306-gcc.3.4.6-ia32_intel64-linux
asia@makai:~/vu/pin-2.11-49306-gcc.3.4.6-ia32_intel64-linux$ ls
LICENSE  README  doc  extras  ia32  intel64  pin  scratch  source  trash
asia@makai:~/vu/pin-2.11-49306-gcc.3.4.6-ia32_intel64-linux$
```

Pin execution

2. Write your own pintool.
   - Numerous examples:

```
asia@makai:~/vu/pin-2.11-49306-gcc.3.4.6-ia32_intel64-linux/source/tools$ ls
AVX            ChildProcess         Debugger            MacTests          MyPinTool
AlignChk       CodeCacheFootprint   GracefulExit        Maid              PapiTools
AttachDetach   CommandLine          I18N                ManualExamples    PinPoints
Buffer         CrossIa32Intel64     Insmix              MemTrace          Probes
CacheClient    DeProf               InstLib             Memory            Replay
CacheFilter    DebugTrace           InstLibExamples     Mix               SegTrace
```

   - Our instruction counting tool

```
asia@makai:~/vu/pintest$ ls
Makefile  icount.cpp
```

Pin execution

3. Set the `PIN_HOME` environment variable to your Pin directory, and `make`.

```
$ pwd
/home/asia/vu/pintest
$ export PIN_HOME=~/vu/pin-2.11-49306-gcc.3.4.6-ia32_intel64-linux/
$ make
g++ -Wall -c -fomit-frame-pointer -std=c++0x -O3 -fno-strict-aliasing -fno-stac
T_IA32 -DHOST_IA32 -DTARGET_LINUX -g  -I. -I/home/asia/vu/pin-2.11-49306-gcc.3.
pin-2.11-49306-gcc.3.4.6-ia32_intel64-linux//source/include/gen -I/home/asia/vu
d2-ia32/include -I/home/asia/vu/pin-2.11-49306-gcc.3.4.6-ia32_intel64-linux//ex
g++ -Wl,--hash-style=sysv -Wl,-Bsymbolic        -shared -Wl,--version-script=/h
//source/include/pintool.ver -L/home/asia/vu/pin-2.11-49306-gcc.3.4.6-ia32_inte
1-49306-gcc.3.4.6-ia32_intel64-linux//ia32/lib -L/home/asia/vu/pin-2.11-49306-g
icount.o -lpin  -lxed -ldwarf -lelf -ldl
```

Pin execution

4. Run ☺

```
$ cd ~/vu/pin-2.11-49306-gcc.3.4.6-ia32_intel64-linux/
$ ./pin -t ~/vu/pintest/icount.so -- /bin/true
Count 98441
```

Demo: Profiling with Pin

Slower Instruction Counting

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

Faster Instruction Counting

Counting at BBL level

```
counter += 3
sub   $0xff, %edx
cmp   %esi, %edx
jle   <L1>
```

```
counter += 2
mov   $0x1, %edi
add   $0x10, %eax
```

Counting at Trace level

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

counter+=3

L1

```
mov   $0x1, %edi
add   $0x10, %eax
counter += 5
```

Example 2: docount++
- instruction counting tool optimized

Instruction counting tool

```
#include “pin.h”
uint64_t icount = 0;

void docount() { icount++; }

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

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(); // never returns
   return 0;
}
```

Instruction counting tool++

```
#include "pin.h"
uint64_t icount = 0;

void PIN_FAST_ANALYSIS_CALL 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(ins, IPOINT_ANYWHERE,
              (AFUNPTR) docount, IARG_FAST_ANALYSIS_CALL,
              IARG_UINT32, BBL_NumIns(bbl), IARG_END);
}

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

int main(int argc, char **argv) {
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();    // never returns
    return 0;
}
```

Direct Pin to call the pintool `Trace` function at the beginning of jitting of each trace.

Instruction counting

A handle to the currently jitted trace.

Use it to iterate through the BBLs of this trace.

```
#include "pin.h"
uint64_t icount = 0;

void PIN_FAST_ANALYSIS_CALL docount(INT32

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

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

int main(int argc, char **argv) {
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();    // never returns
    return 0;
}
```

Instruction counting tool++

```
#include "pin.h"
uint64_t icount = 0;

void PIN_FAST_ANALYSIS_CALL 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(ins, IPOINT_ANYWHERE,
               (AFUNPTR) docount, IARG_FAST_ANALYSIS_CALL,
               IARG_UINT32, BBL_NumIns(bbl), IARG_END);
}

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

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();    // never returns
   return 0;
}
```

Call **docount** before executing each BBL.

Pass an arg of type `IARG_UINT32`, and value `BBL_NumIns(bbl)`.

Instruction counting tool++

```
#include "pin.h"
uint64_t icount = 0;

void PIN_FAST_ANALYSIS_CALL 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(ins, IPOINT_ANYWHERE,
              (AFUNPTR) docount, IARG_FAST_
              IARG_UINT32, BBL_NumIns(bbl), IARG_
}

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

int main(int argc, char **argv) {
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();    // never returns
    return 0;
}
```

Insert the instrumentation anywhere in the BBL – this might enable Pin find an optimal place .

References

- The official Pin webpage
  - www.pintool.org
- User's Manual
  - https://software.intel.com/sites/landingpage/pintool/docs/67254/Pin/html/
  - A lot of examples!
  - Debugging tips ☺
- Pin User Group (PinHeads)
  - http://tech.groups.yahoo.com/group/pinheads/
  - Pin users and Pin developers answer questions