



# Pin Optimizations

**TA: Nathan Beckmann**

*Adapted from: Owen Chin 2012, Tushar Krishna  
2011, and Intel's Tutorial at CGO 2010*

*From the last tutorial...*

# What is Instrumentation?



- Instrumentation is a technique that inserts extra code into a program to collect runtime information
- PIN does dynamic binary instrumentation

Runtime

No need to  
re-compile  
or re-link

# Instrumentation: Instruction Count



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

Analysis routine

Instrumentation routine

```
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**
  - ☞ **Occurs *first time* an instruction is executed**
- **Analysis routines** define what to do when instrumentation is **activated**
  - ☞ **Occurs *every time* an instruction is executed**

# 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

# 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

# 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

- Instrumentation with Pin can be done at 3 different granularities:

- Instruction

- Basic block

- A sequence of instructions (with at most one unconditional control flow instruction)
- Single entry, single exit

- Trace

- A sequence of basic blocks (with at most one changing instruction)
- Single entry, multiple exits

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

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

w



# Instrumentation Granularity

- Instrumentation with Pin can be done at 3 different granularities:

- Instruction

- Basic block

- A sequence of instructions (with at most one unconditional control flow instruction)
- Single entry, single exit

- Trace

- A sequence of basic blocks (with at most one changing instruction)
- Single entry, multiple exits

6 insts

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

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

w



# Instrumentation Granularity

- Instrumentation with Pin can be done at 3 different granularities:

- Instruction

- Basic block

- A sequence of instructions (with at most one unconditional control transfer)
- Single entry, single exit

- Trace

- A sequence of basic blocks (with at most one changing instruction)
- Single entry, multiple exits

6 insts, 2 basic blocks

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

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

w



# Instrumentation Granularity

- Instrumentation with Pin can be done at 3 different granularities:

- Instruction

- Basic block

- A sequence of instructions (with at most one unconditional control flow instruction)
- Single entry, single exit

- Trace

- A sequence of basic blocks (with at most one changing instruction)
- Single entry, multiple exits

6 insts, 2 basic blocks, 1 trace

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

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

w

# 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
```

# Recap of Pintool 1: Instruction Count



```
counter++;  
sub  $0xff, %edx
```

- Straightforward, but the counting can be more efficient

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

# Pintool 3: Faster Instruction Count

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

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

basic blocks (bbl)





```
#include <stdio.h>
#include "pin.H"
UINT64 icount = 0;
```

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

*analysis routine*

```
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);
    }
}
```

*instrumentation routine*

```
void Fini(INT32 code, void *v) {
    fprintf(stderr, "Count %lld\n", icount);
}
```

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



# Reducing Frequency of Calling Analysis Routines



- Key:
  - Instrument at the largest granularity whenever possible:
    - Trace > Basic Block > Instruction

# 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

# Reducing Pintool's Overhead



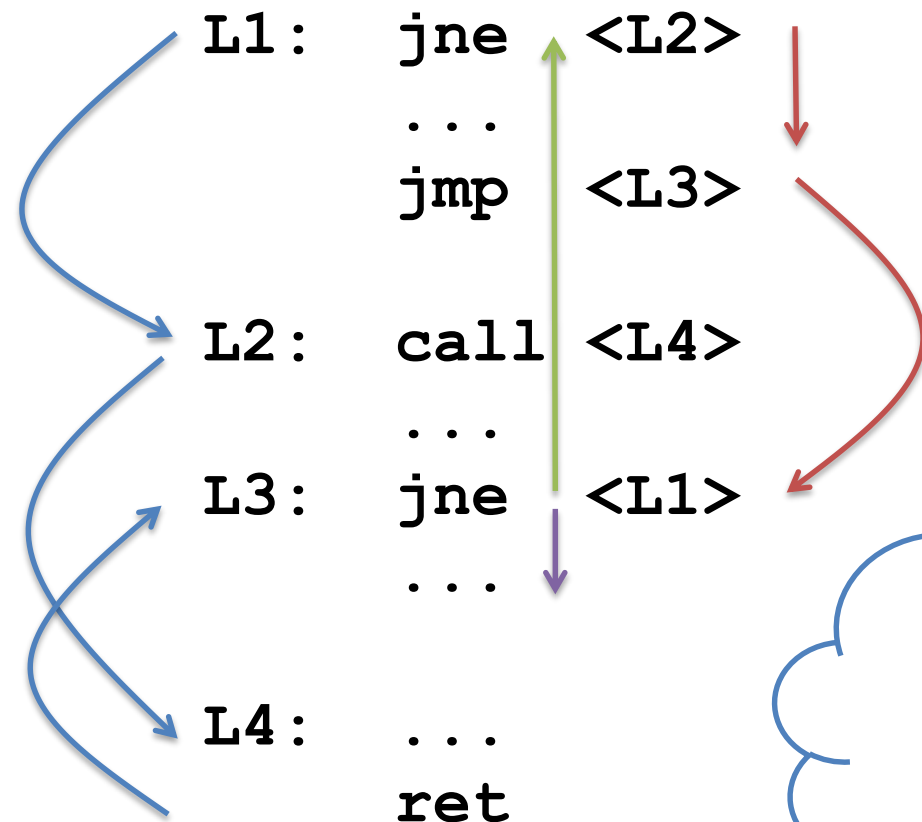
## Pintool's Overhead

Instrumentation Routines Overhead + Analysis Routines Overhead

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

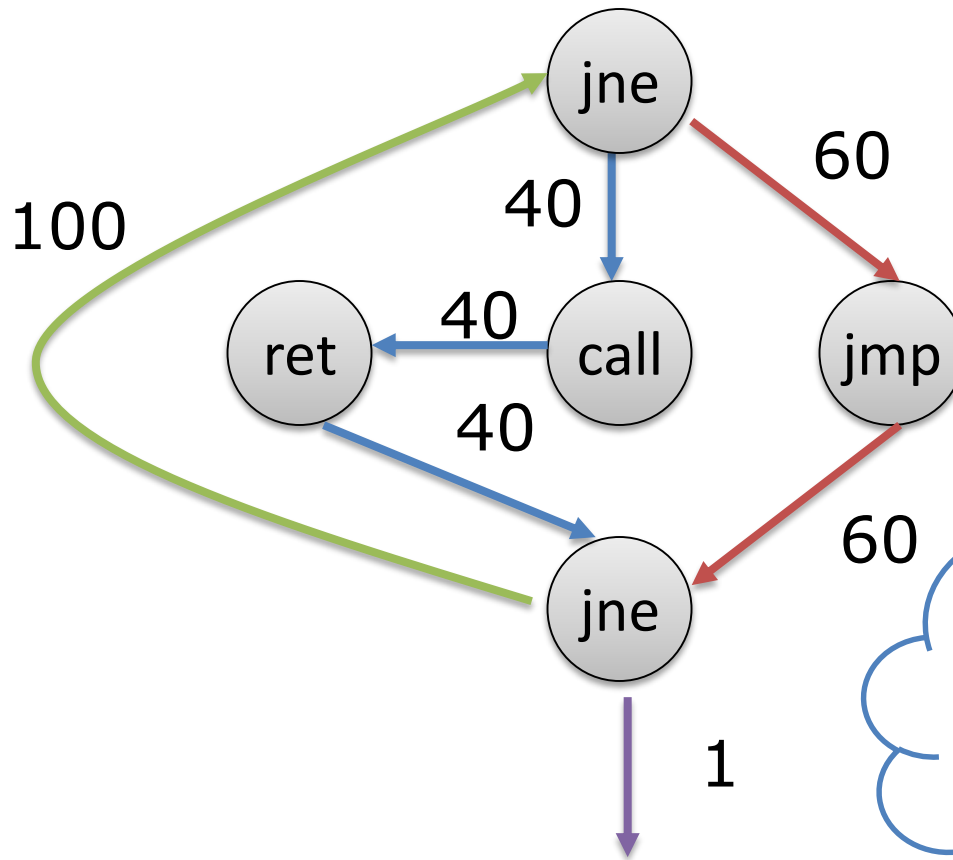
Work required for transiting to Analysis Routine + Work done inside Analysis Routine

# Example: Counting Control Flow Edges



How often is each branch taken?

# Example: Counting Control Flow Edges



How often is each branch taken?

# Edge Counting: a Slower Version



```
...  
void docount2(ADDRINT src, ADDRINT dst, INT32 taken)  
{  
    COUNTER *pedg = Lookup(src, dst);  
    pedg->count += taken;  
}
```

```
void Instruction(INS ins, void *v) {  
    if (INS_IsBranchOrCall(ins))  
    {  
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount2,  
            IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,  
            IARG_BRANCH_TAKEN, IARG_END);  
    }  
}
```

1 if taken, 0 if not taken

# Inefficiency in Program



- About every 5th instruction executed in a typical application is a branch.
- Lookup will be called whenever these instructions are executed
  - significant application slowdown
- **Direct vs. Indirect Branches**
  - Branch Address in instruction vs. Branch Address in Register
  - Static vs. Dynamic

# Edge Counting: a Faster Version



```
void docount(COUNTER* pedg, INT32 taken) {
    pedg->count += taken;
}
```

```
void docount2(ADDRINT src, ADDRINT dst, INT32 taken) {
    COUNTER *pedg = Lookup(src, dst);
    pedg->count += taken;
}
```

```
void Instruction(INS ins, void *v) {
    if (INS_IsDirectBranchOrCall(ins)) {
        COUNTER *pedg = Lookup(INS_Address(ins),
                               INS_DirectBranchOrCallTargetAddress(ins));
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount,
                       IARG_ADDRINT, pedg, IARG_BRANCH_TAKEN, IARG_END);
    } else
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount2,
                       IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
                       IARG_BRANCH_TAKEN, IARG_END);
}
```



# Why didn't I use "if"?



```
void docount2(ADDRINT src, ADDRINT dst, INT32 taken) {
    if (!taken)
        return;

    COUNTER *pedg = Lookup(src, dst);
    pedg->count++;
}
```

*VS.*

```
void docount2(ADDRINT src, ADDRINT dst, INT32 taken) {
    COUNTER *pedg = Lookup(src, dst);
    pedg->count += taken;
}
```

**Can be inlined by Pin**

# Reducing Work Done in Analysis Routines



- Key:
  - Shifting computation from Analysis Routines to Instrumentation Routines whenever possible

# Some other optimizations...



- Reduce the number of arguments to analysis routine.
  - For example, instead of passing TRUE/FALSE, create 2 analysis functions.
- If an instrumentation can be inserted anywhere in a basic block:
  - Let Pin know via **IPOINT\_ANYWHERE**
  - Pin will find the best point to insert the instrumentation to minimize register spilling

# Takeaways..



- Reduce **frequency** of calling analysis routines by instrumenting at **the largest granularity** whenever possible
- Reduce **the amount of work** done in analysis routines by **shifting computation** from Analysis Routines to Instrumentation Routines whenever possible