6.823
Pin Optimizations

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

From the Video 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

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
```

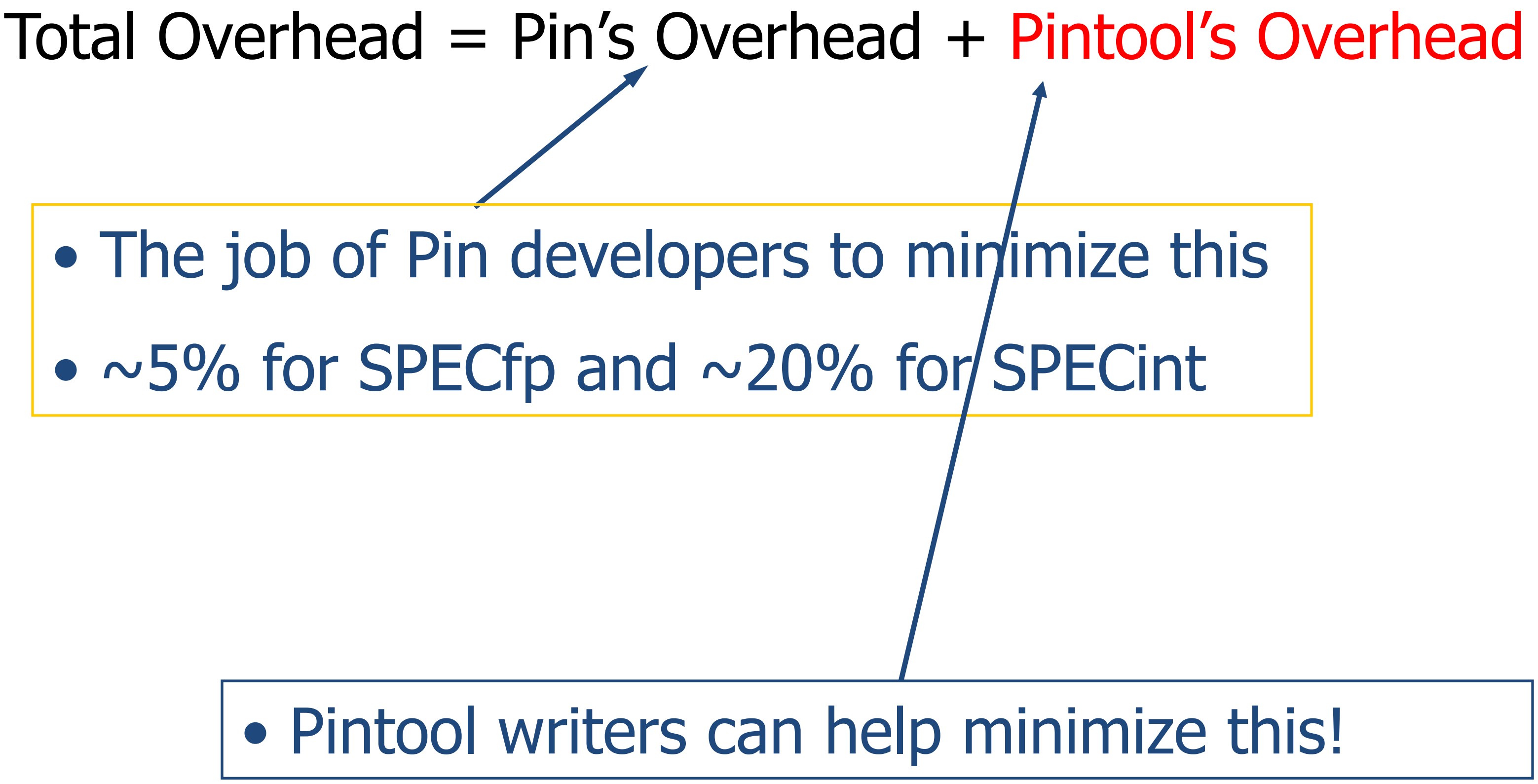
Instrumentation vs. Analysis

- **Instrumentation routines** define where instrumentation is **inserted**
  - ☞ **Occurs immediately before an instruction is executed for the first time.**

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

How to Write Efficient Pintools

Reducing Instrumentation Overhead

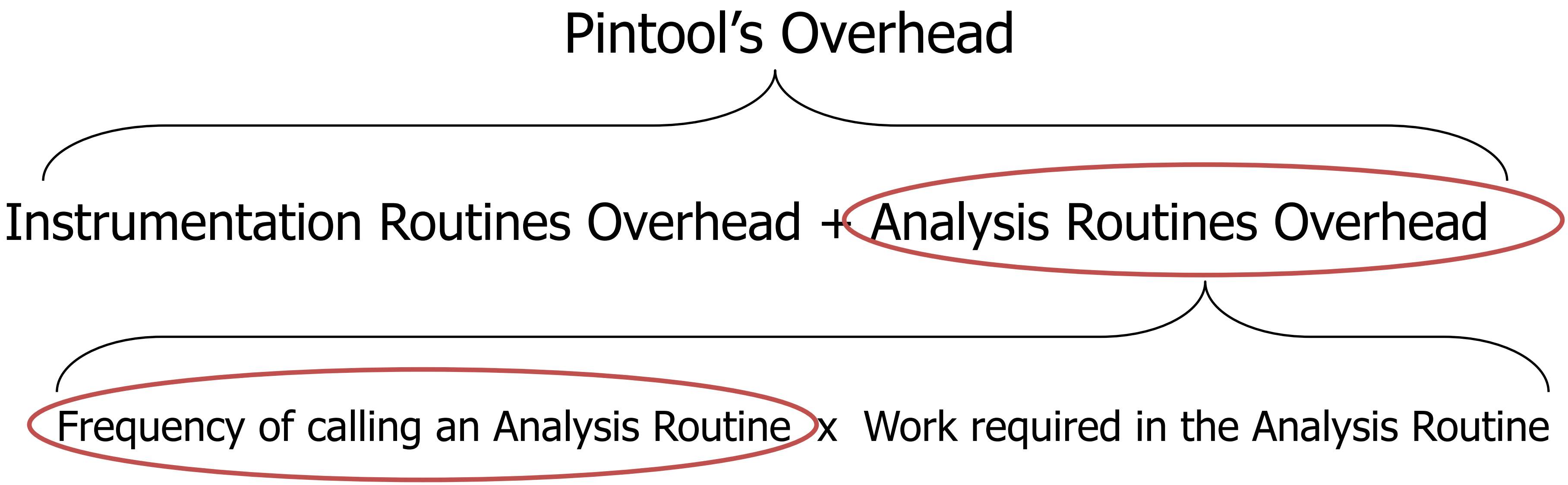


Total Overhead = Pin’s Overhead + Pintool’s Overhead

- 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

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 entrance, single exit
  - Trace
    - A sequence of basic blocks terminated at an unconditional control-flow changing instruction
    - Single entrance, multiple exits

Instrumentation Granularity

- Instrumentation with Pin can be done at 3 different granularities:
  - Instruction
  - Basic block
    - A sequence of instruct unconditional) control
    - Single entrance, single
  - Trace
    - A sequence of basic bl w changing instruction
    - Single entrance, multiple exits

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

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

Instrumentation Granularity

- Instrumentation with Pin can be done at 3 different granularities:
  - Instruction
  - Basic block
    - A sequence of instruct
      unconditional) control
    - Single entrance, single
  - Trace
    - A sequence of basic bl w
      changing instruction
    - Single entrance, multiple exits

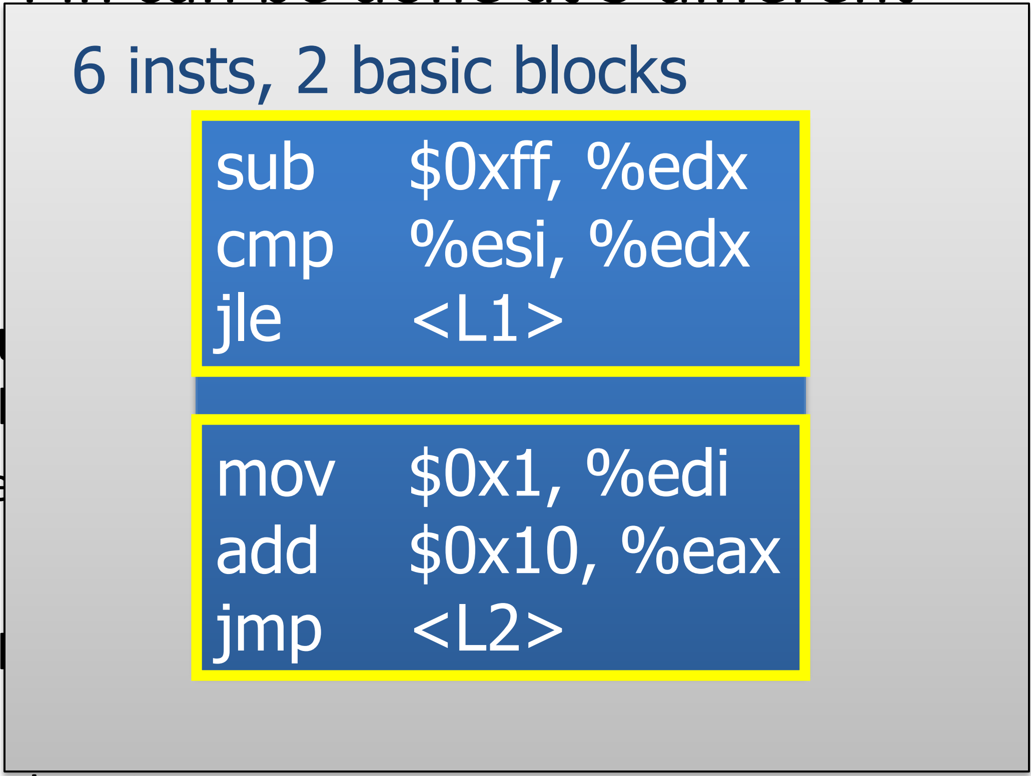
6 insts

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

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

Instrumentation Granularity

- Instrumentation with Pin can be done at 3 different granularities:
  - Instruction
  - Basic block
    - A sequence of instruct unconditional) control
    - Single entrance, single
  - Trace
    - A sequence of basic bl w changing instruction
    - Single entrance, multiple exits



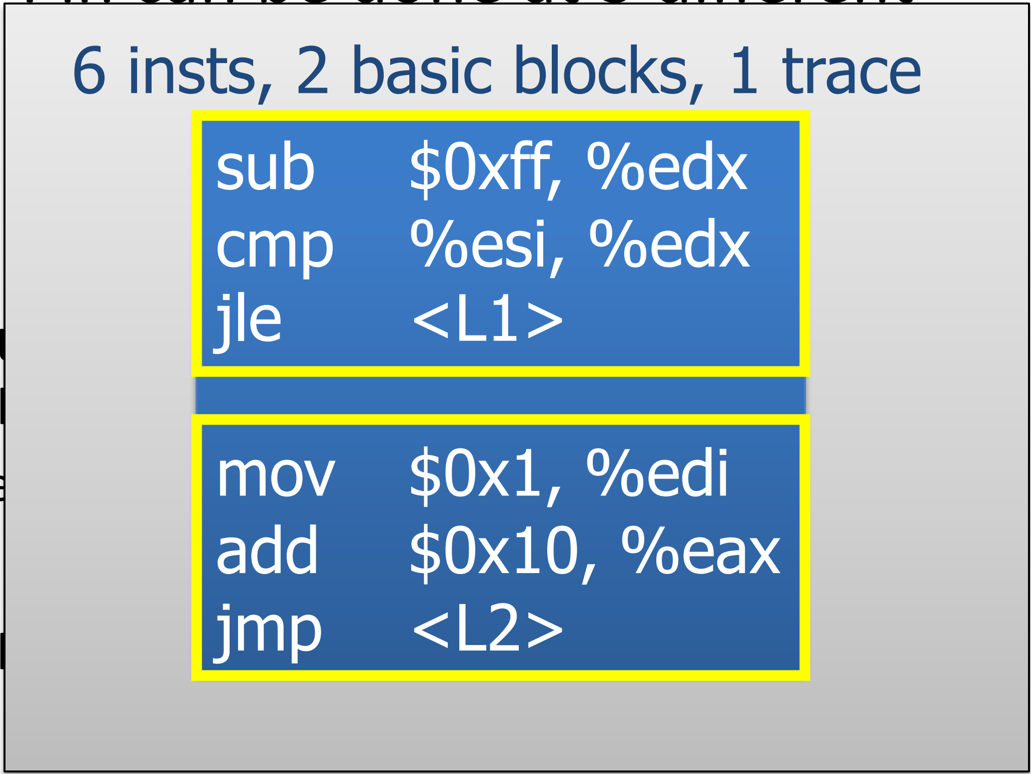
6 insts, 2 basic blocks

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

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

Instrumentation Granularity

- Instrumentation with Pin can be done at 3 different granularities:
  - Instruction
  - Basic block
    - A sequence of instruct
      unconditional) control
    - Single entrance, single
  - Trace
    - A sequence of basic bl   w
      changing instruction
    - Single entrance, 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>
```

Recap of Pintool: Instruction Count

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

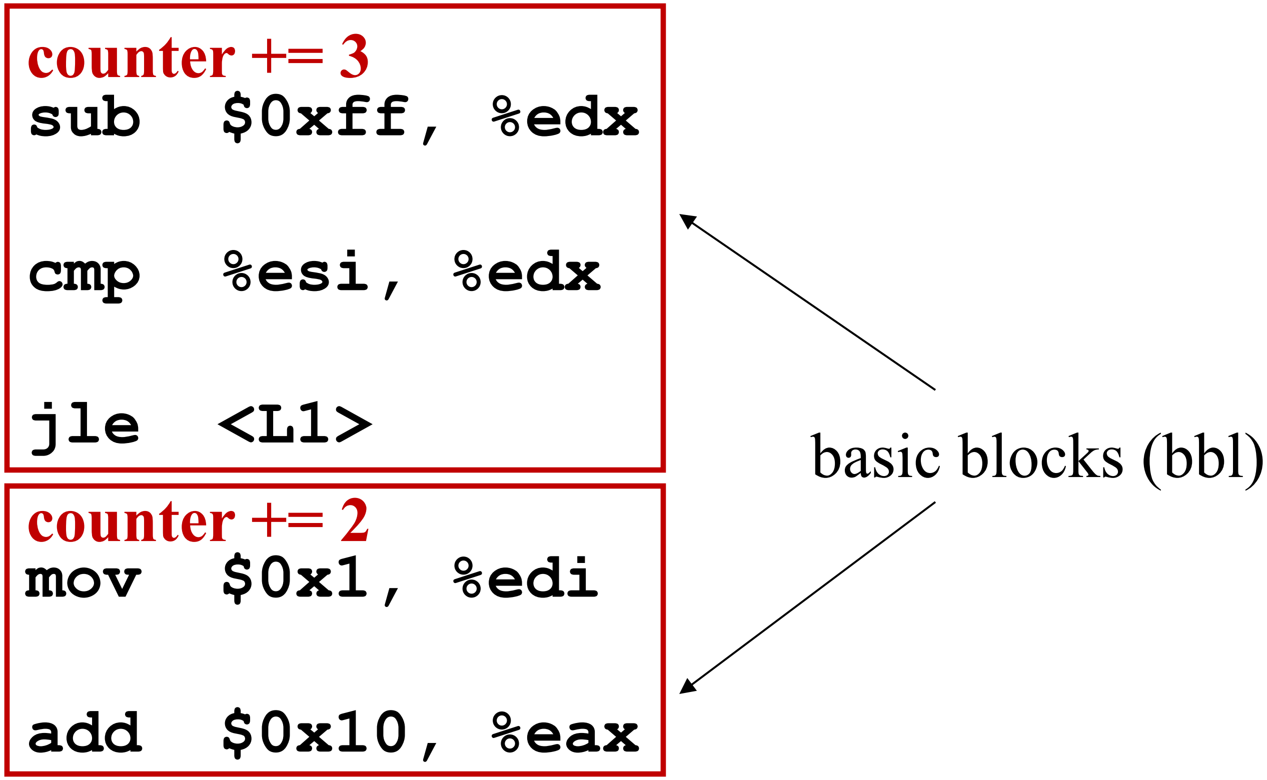
Recap of Pintool: Instruction Count

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

- Straightforward, but the counting can be more efficient

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

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

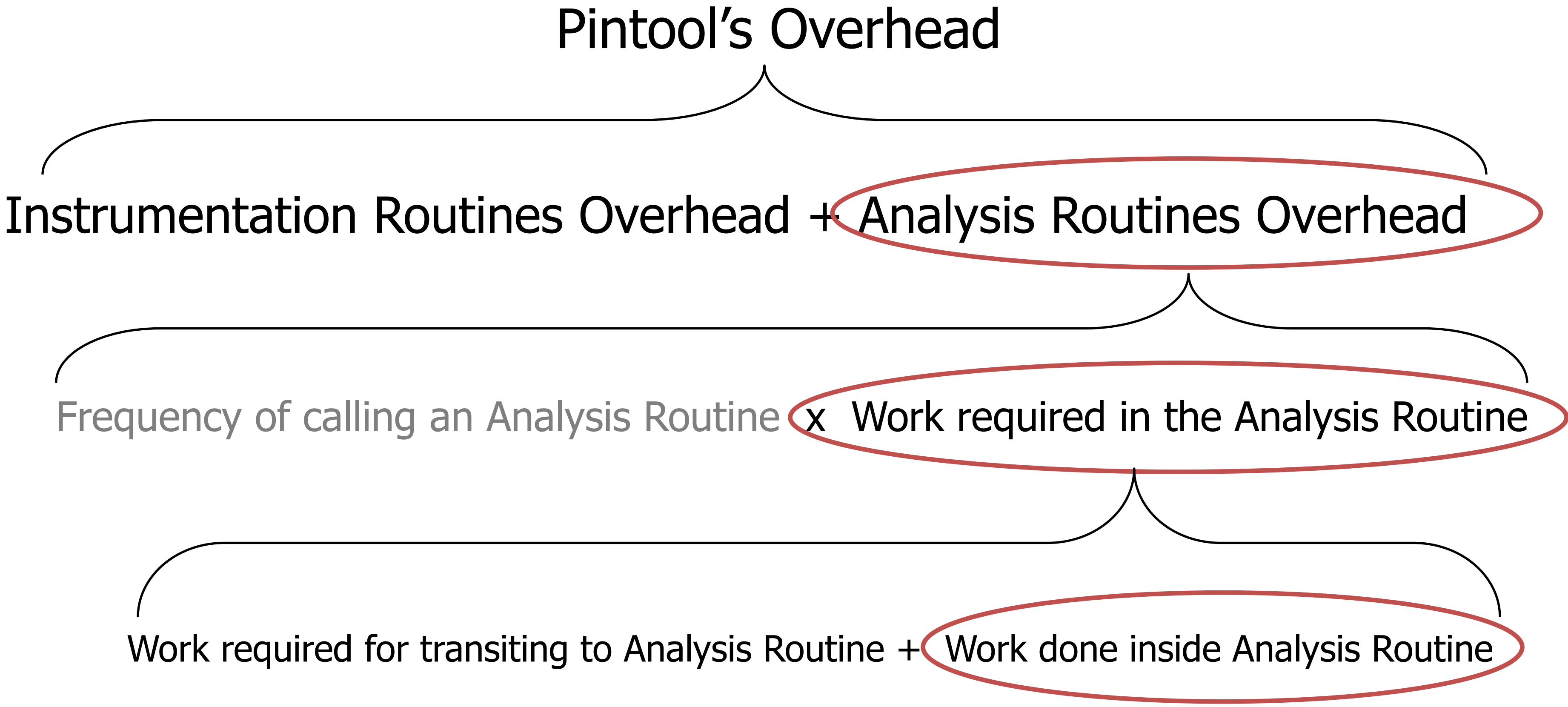
Reducing Pintool's Overhead

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



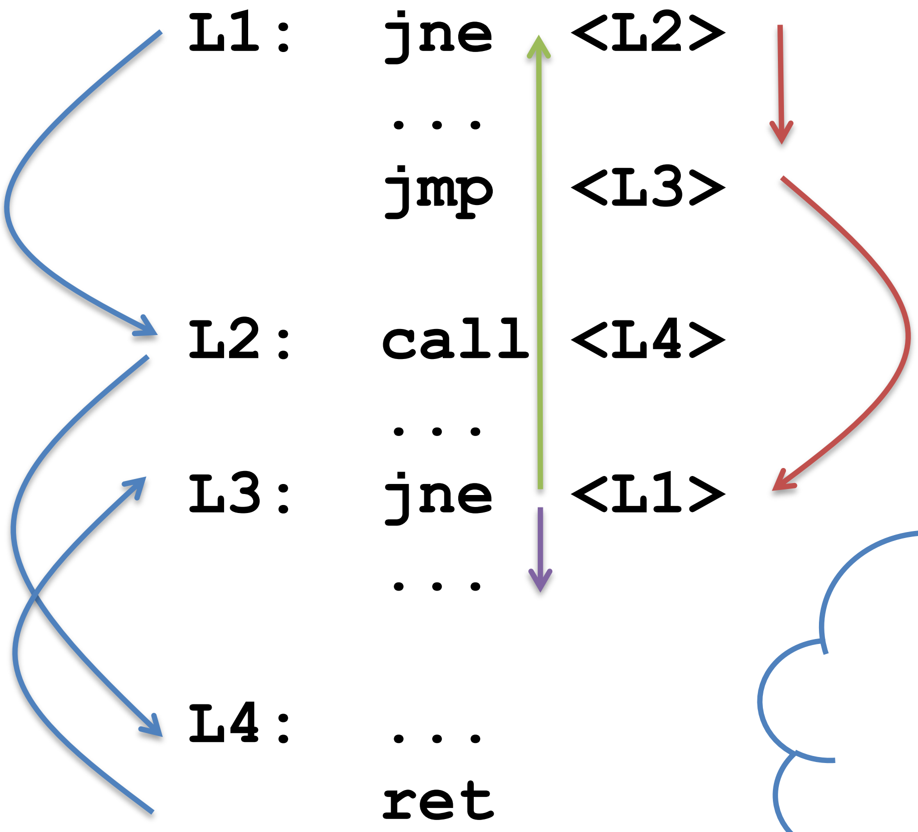
Pintool's Overhead

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



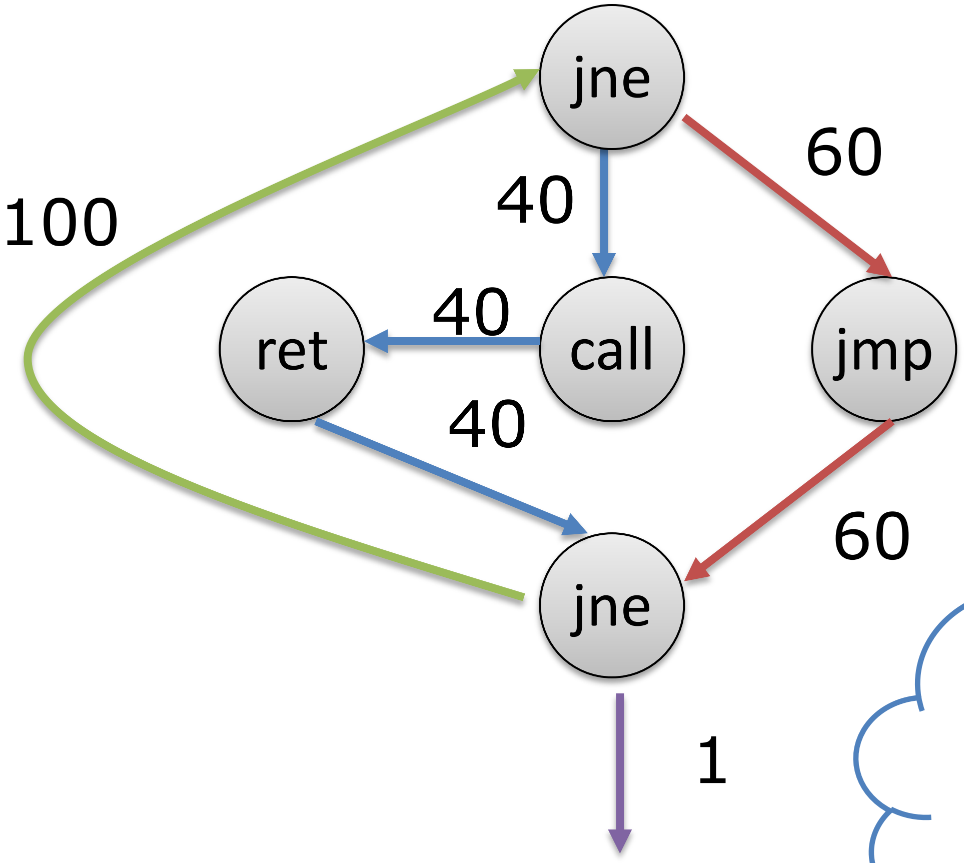
```
L1:   jne   <L2>
      ...
      jmp   <L3>

L2:   call  <L4>
      ...
L3:   jne   <L1>
      ...

L4:   ...
      ret
```

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.
- Edge lookup will be called whenever these instruction 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* pedge, 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 if (INS_IsBranchOrCall(ins))
       INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount2,
                      IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
                      IARG_BRANCH_TAKEN, IARG_END);
}
…
```

Eliminating Control Flow

```
void docount(COUNTER* pedge, INT32 taken)
{
   if (!taken)
       return;
   pedg->count++;
}
```

vs.

```
void docount(COUNTER* pedge, INT32 taken)
{
   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 (used in BBL_InsertCall())
  - Pin will find the best point to insert the instrumentation to minimize register spilling

Takeaways..

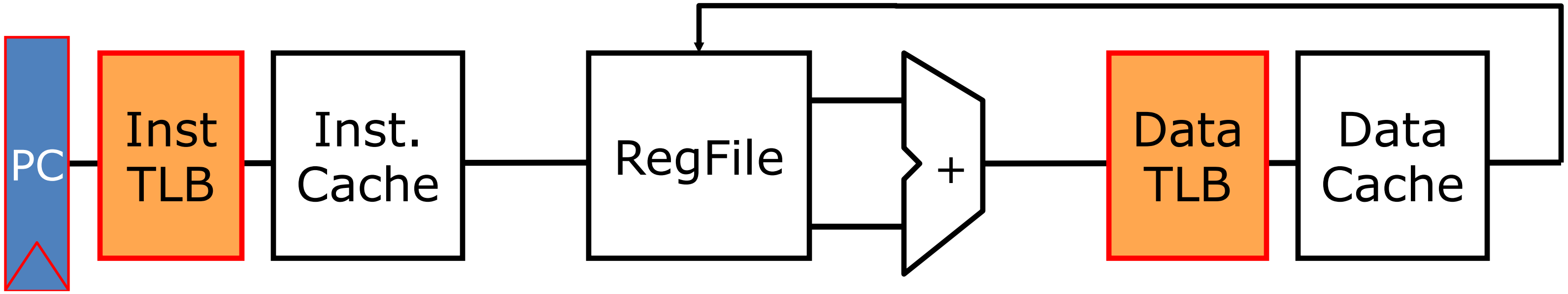
- Reduce frequency of calling analysis routines by instrumenting at **the largest granularity** whenever possible

- Reduce the amount of work done in analysis routines by **shifting computation** from Analysis Routines to Instrumentation Routines whenever possible

Lab 1 due in a week

- Design 3 different types of caches
  - Virtually Indexed, Virtually Tagged
  - Physically Indexed, Physically Tagged
  - Virtually Indexed, Physically Tagged
- Caches and Virtual Memory covered in Lectures 2-4
- Remember to start early!

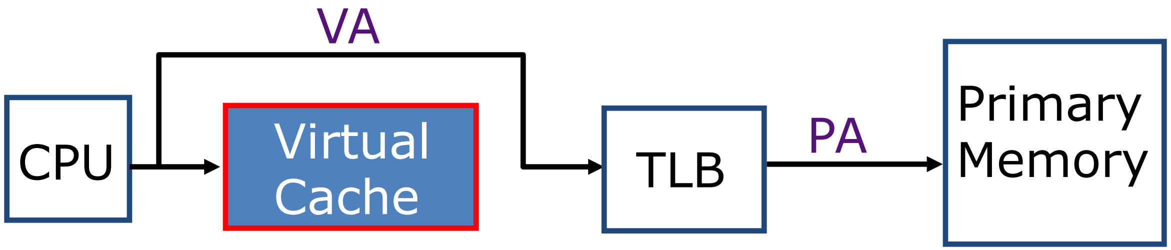
Caches & Virtual Memory



- Processor works with virtual addresses
  - If we grab the index and tag from the physical address, need address translation -> TLB access
  - Avoid this: virtually-addressed cache

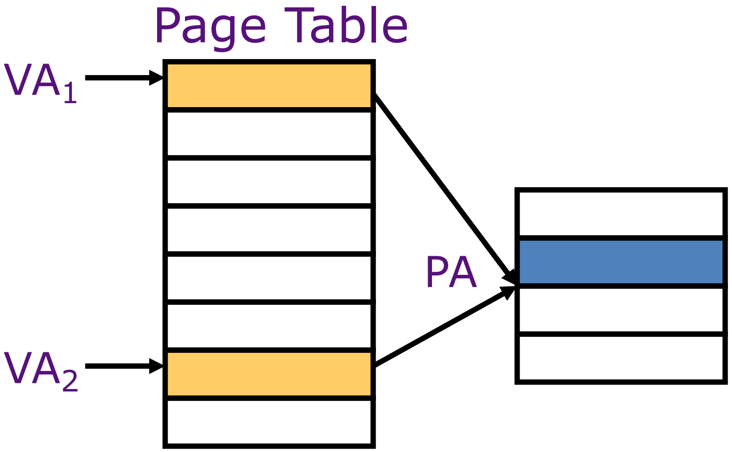
Caches & Virtual Memory

Alternative: place the cache before the TLB



- Now, cache hits are fast
- Problem 1: Consider VA_1(from process 1) and VA_2(from process 2)
  - What if we context switch, and $VA_1 == VA_2$?

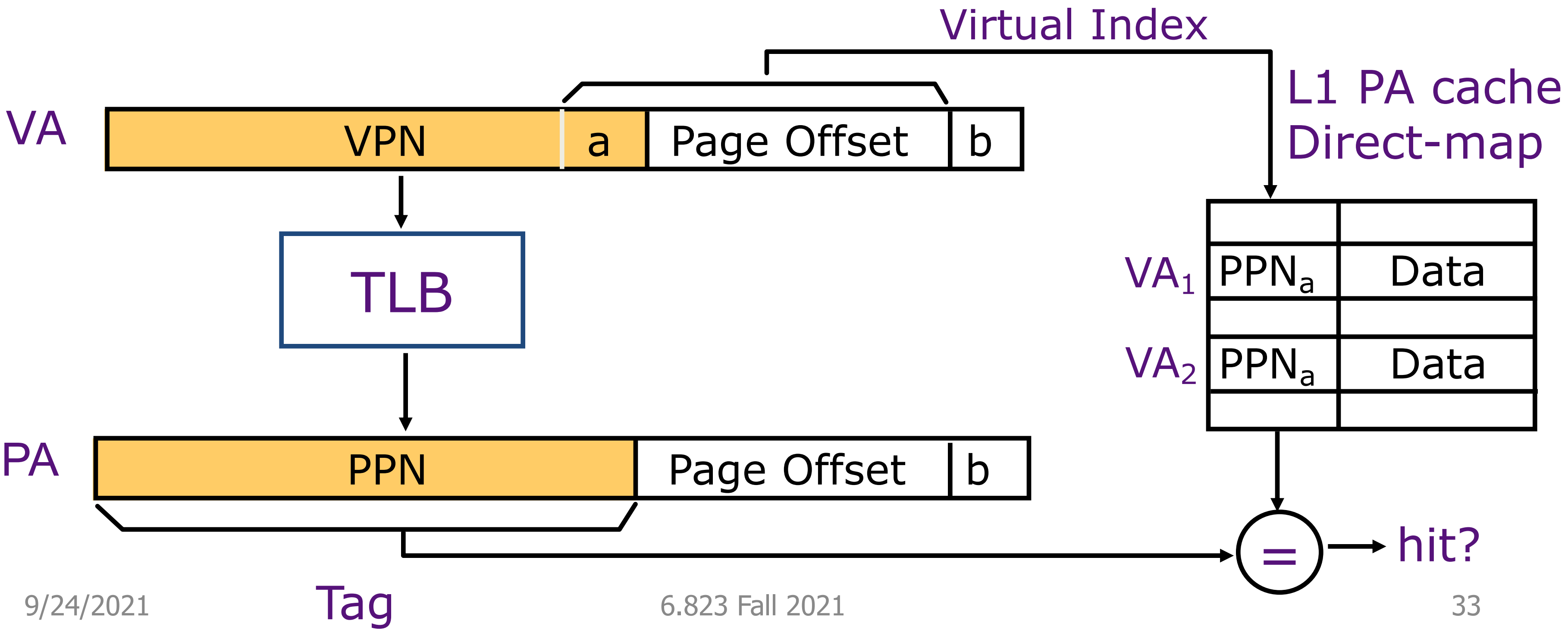
Caches & Virtual Memory



- Problem 2: Consider VA_1, VA_2 (not necessarily from different processes)
  - What if VA_1 != VA_2, but they map to the same physical address?

Caches & Virtual Memory

- Intuition: Physical tags solve Problem 1
- Solves Problem 2 as long as the index bits are the same between VA_1 and VA_2



Tips

- Ask questions on Piazza.

- ssh <athenausername>@vlsifarm-0*X*.mit.edu or
- ssh <athenausername>@eecs-ath-4*X*.mit.edu
  - eecs-ath-4*X* machines are **much** more powerful

- Suggested reading on caches and virtual memory on the course website.