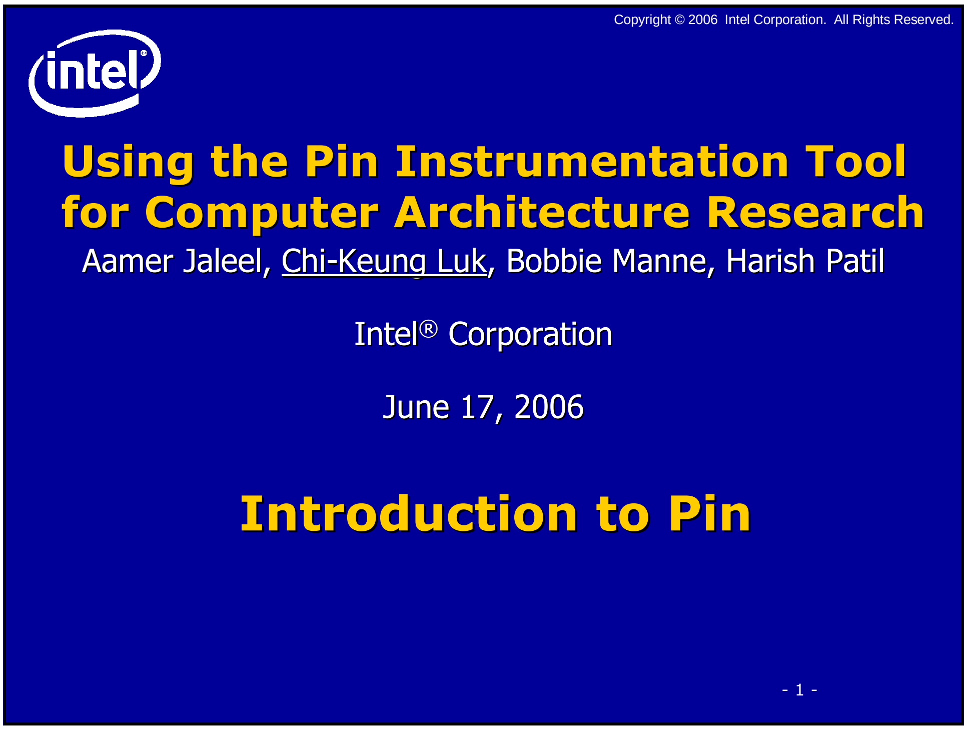


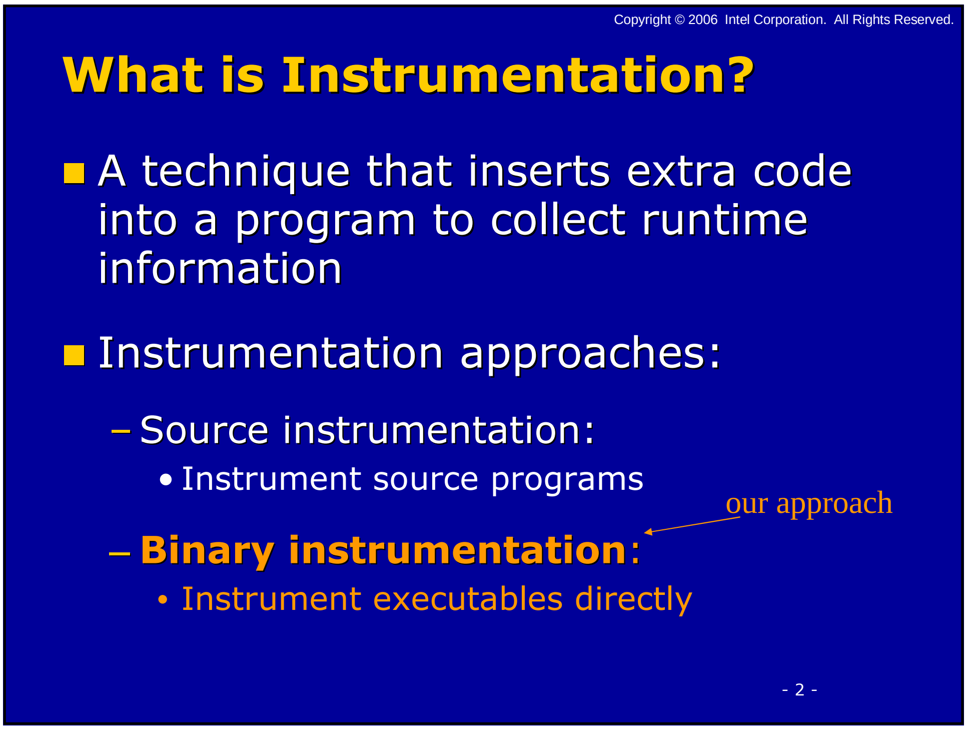
Using the Pin Instrumentation Tool for Computer Architecture Research

Aamer Jaleel, Chi-Keung Luk, Bobbie Manne, Harish Patil

Intel® Corporation

June 17, 2006

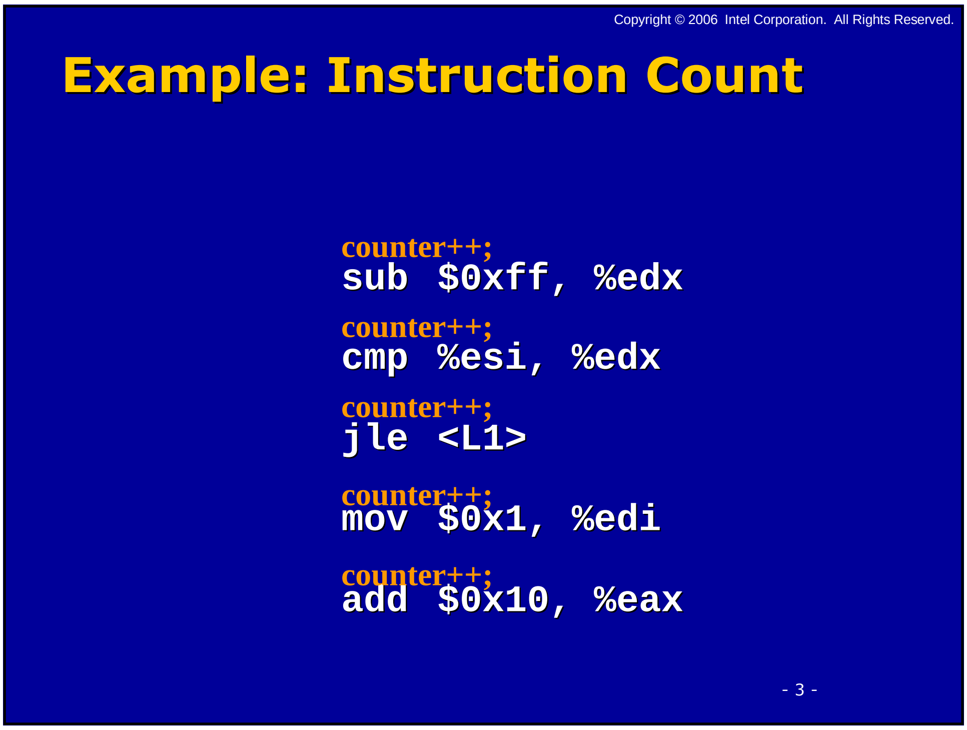
Introduction to Pin



What is Instrumentation?

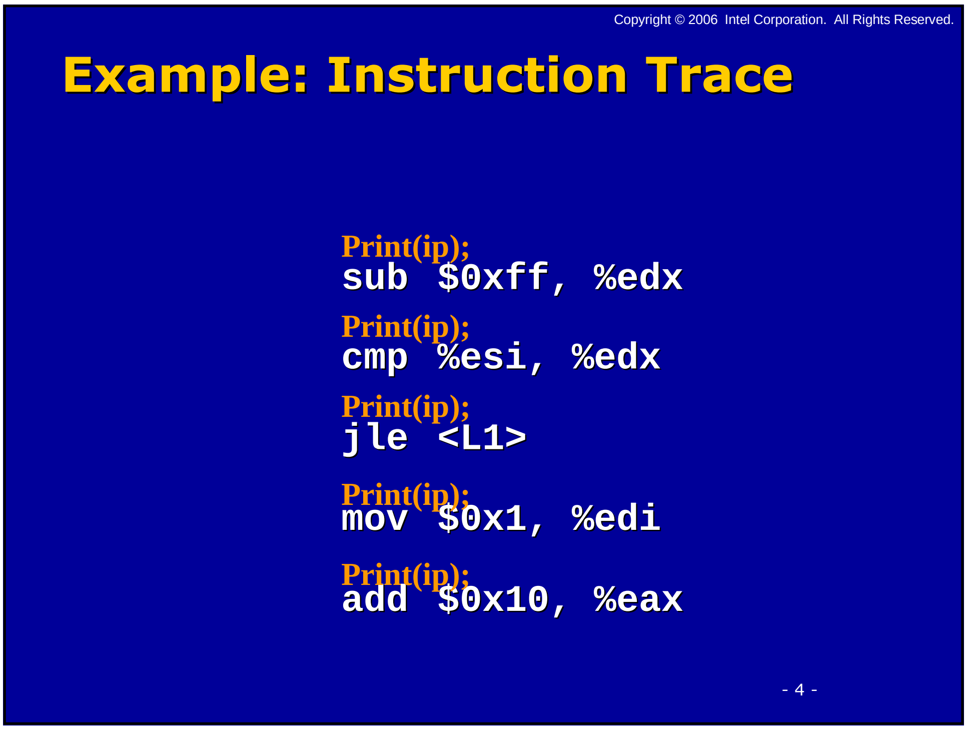
- A technique that inserts extra code into a program to collect runtime information
- Instrumentation approaches:
  - Source instrumentation:
    - Instrument source programs
  - **Binary instrumentation**:
    - Instrument executables directly

our approach



Example: Instruction Count

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



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)

How is Instrumentation used in Computer Architecture?

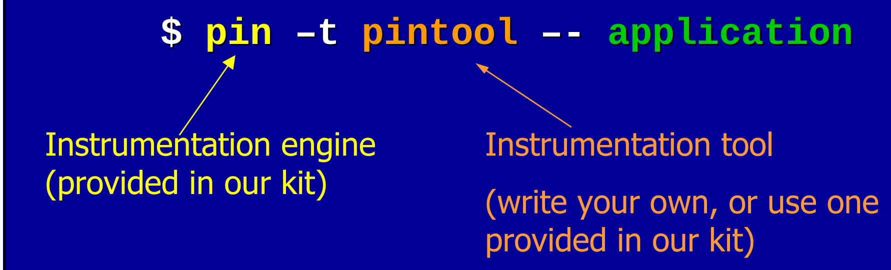
- Trace Generation
- Branch Predictor and Cache Modeling
- Fault Tolerance Study
- Emulating Speculation
- Emulating New Instructions

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
- 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
```

Instrumentation engine (provided in our kit)

Instrumentation tool

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

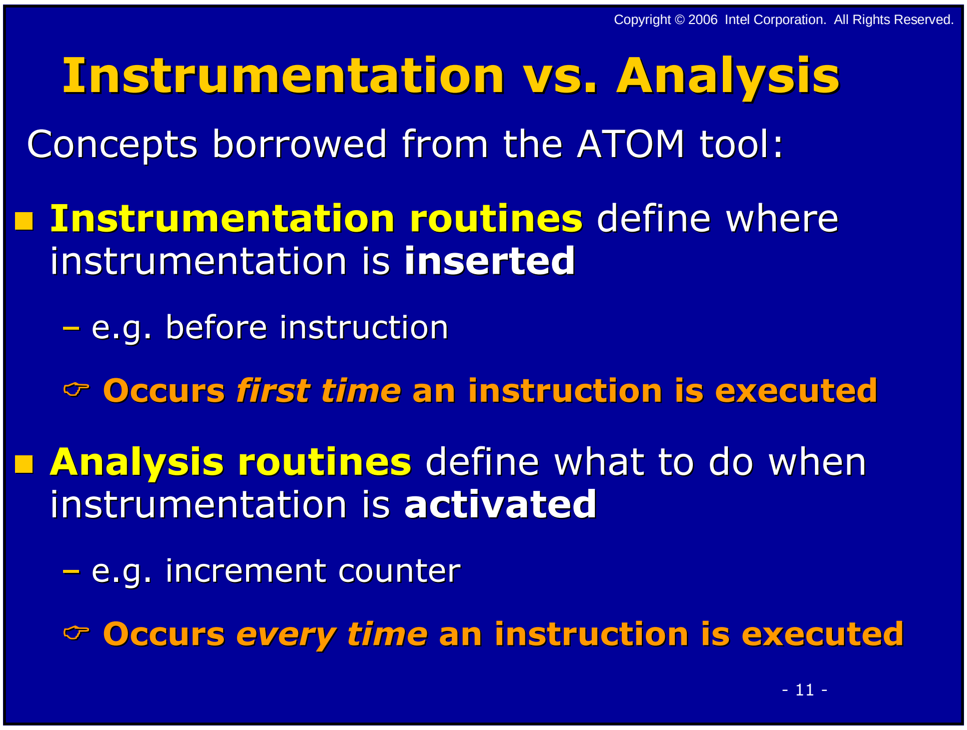
- Attach to and instrument an application

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

Writing Pintools

Pin Instrumentation APIs

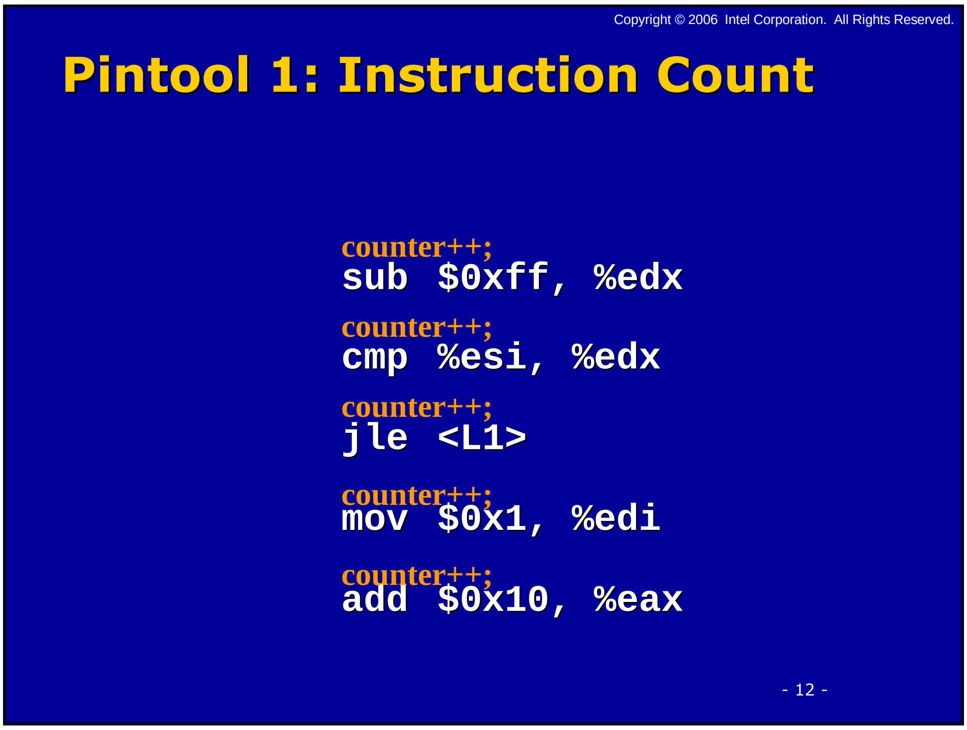
- Basic APIs are architecture independent:
  - Provide common functionalities like determining:
    - Control-flow changes
    - Memory accesses
- Architecture-specific APIs
  - E.g., Info about segmentation registers on IA32
- Call-based APIs:
  - Instrumentation routines
  - Analysis routines



Instrumentation vs. Analysis

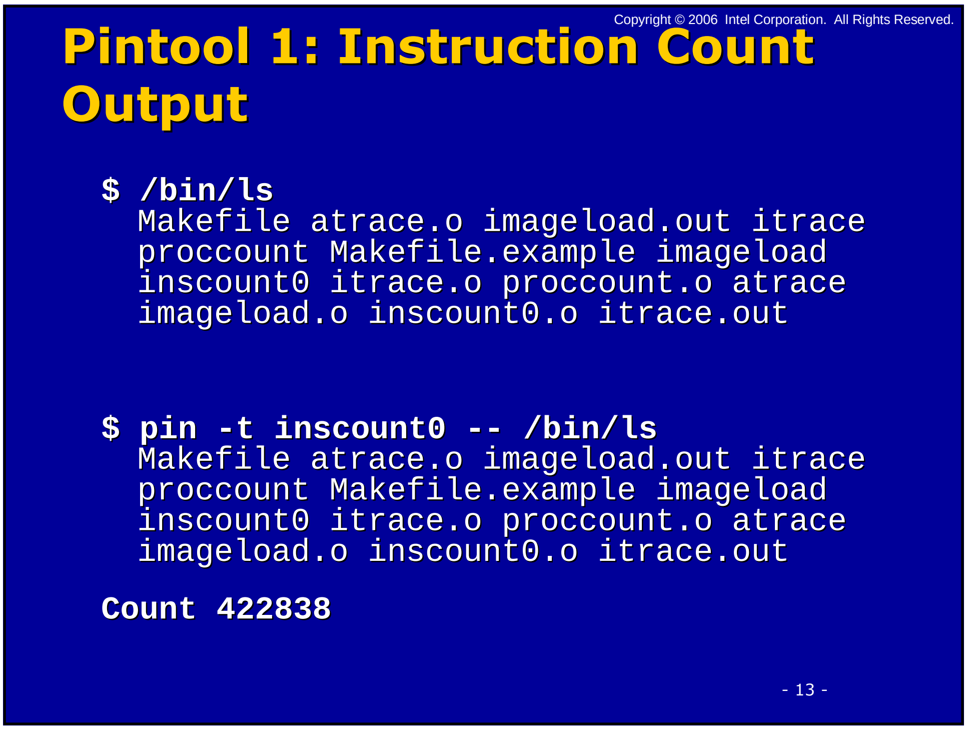
Concepts borrowed from the ATOM tool:

- **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**



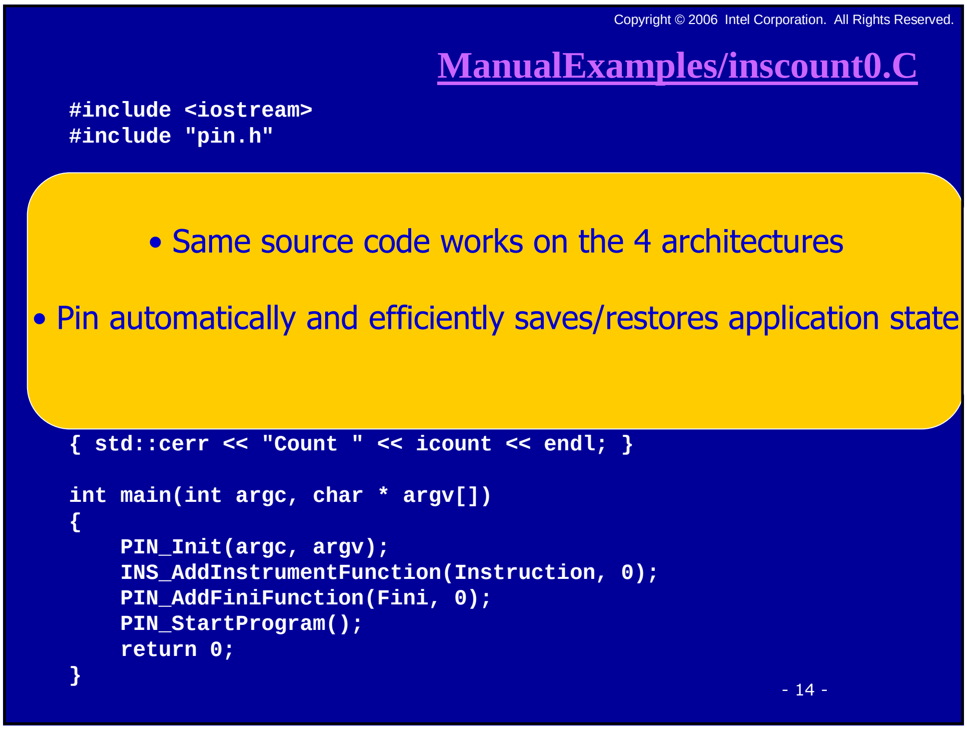
Pintool 1: Instruction Count

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



Pintool 1: Instruction Count Output

```
$ /bin/ls
  Makefile atrace.o imageload.out itrace
  proccount Makefile.example imageload
  inscount0 itrace.o proccount.o atrace
  imageload.o inscount0.o itrace.out


$ pin -t inscount0 -- /bin/ls
  Makefile atrace.o imageload.out itrace
  proccount Makefile.example imageload
  inscount0 itrace.o proccount.o atrace
  imageload.o inscount0.o itrace.out

Count 422838
```



ManualExamples/inscount0.C

```
#include <iostream>
#include "pin.h"
```

- Same source code works on the 4 architectures
- Pin automatically and efficiently saves/restores application state

```
{ 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;
}
```

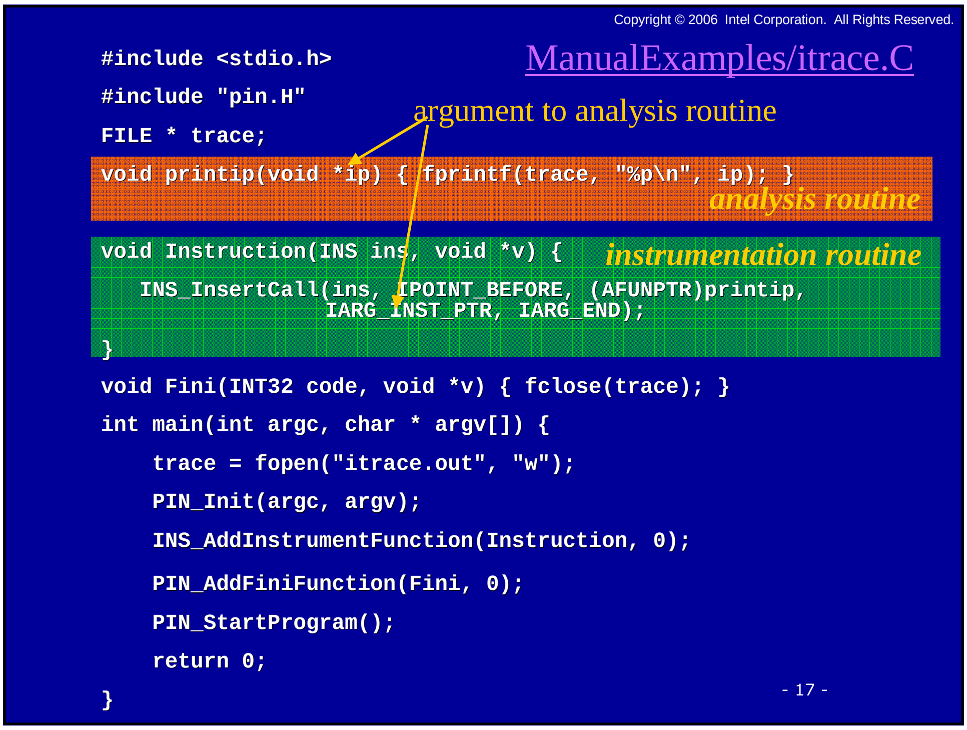
Pintool 2: Instruction Trace

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

```
Print(ip);
mov  $0x1, %edi
Print(ip);
add  $0x10, %eax
```

Pintool 2: Instruction Trace Output

```
$ pin -t itrace -- /bin/ls
  Makefile atrace.o imageload.out itrace
  proccount Makefile.example imageload
  inscount0 itrace.o proccount.o atrace
  imageload.o inscount0.o itrace.out


$ head -4 itrace.out

0x40001e90

0x40001e91

0x40001ee4

0x40001ee5
```



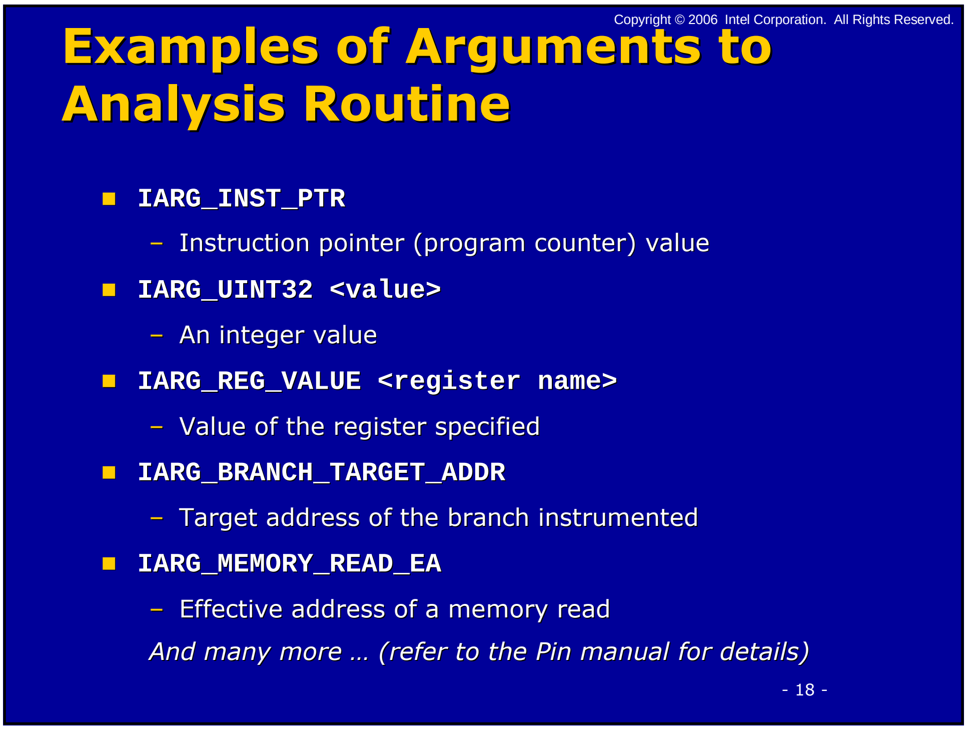
ManualExamples/itrace.C

argument to analysis routine

analysis routine

instrumentation routine

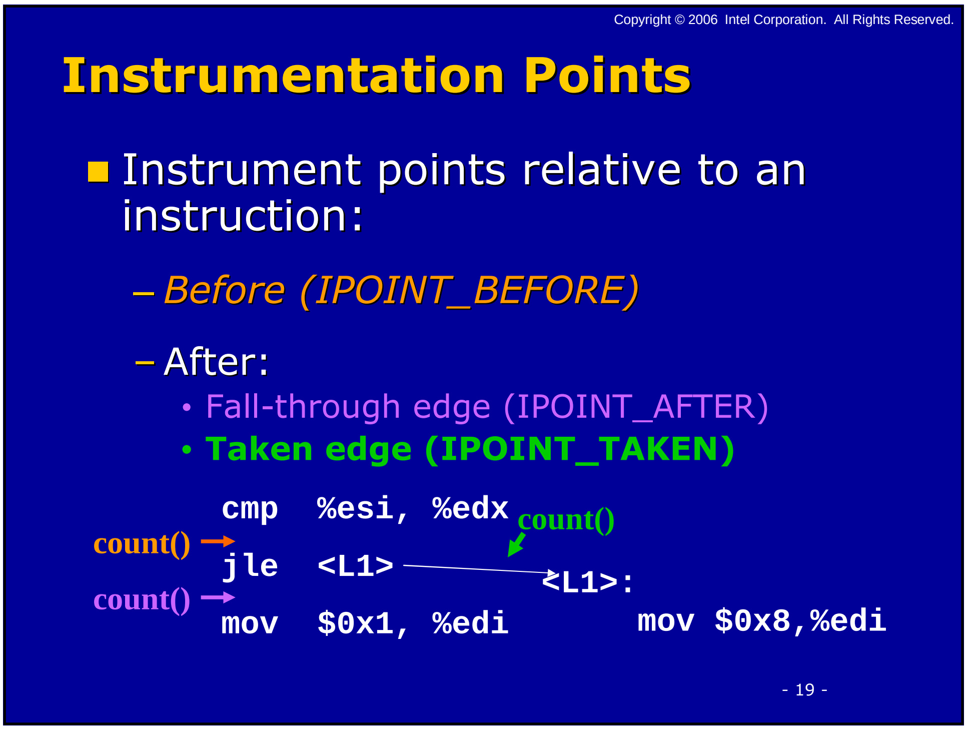
```
#include <stdio.h>
#include "pin.H"
FILE * trace;
void printip(void *ip) { fprintf(trace, "%p\n", ip); }
void Instruction(INS ins, void *v) {
   INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip,
                 IARG_INST_PTR, IARG_END);
}
void Fini(INT32 code, void *v) { fclose(trace); }
int 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;
}
```



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

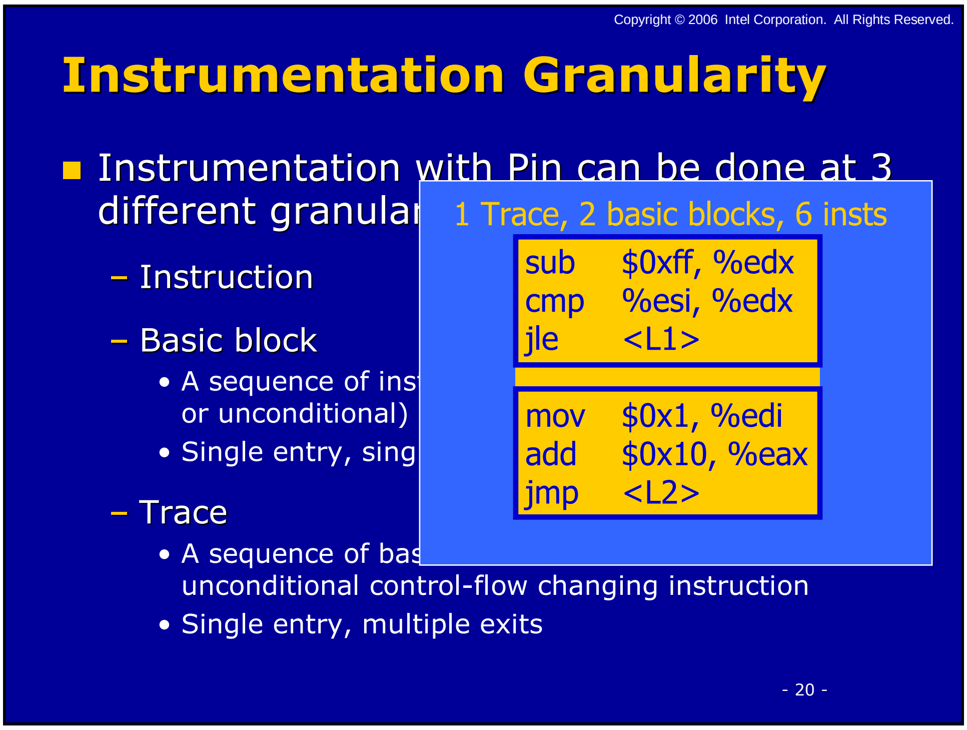
And many more … (refer to the Pin manual for details)



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
```

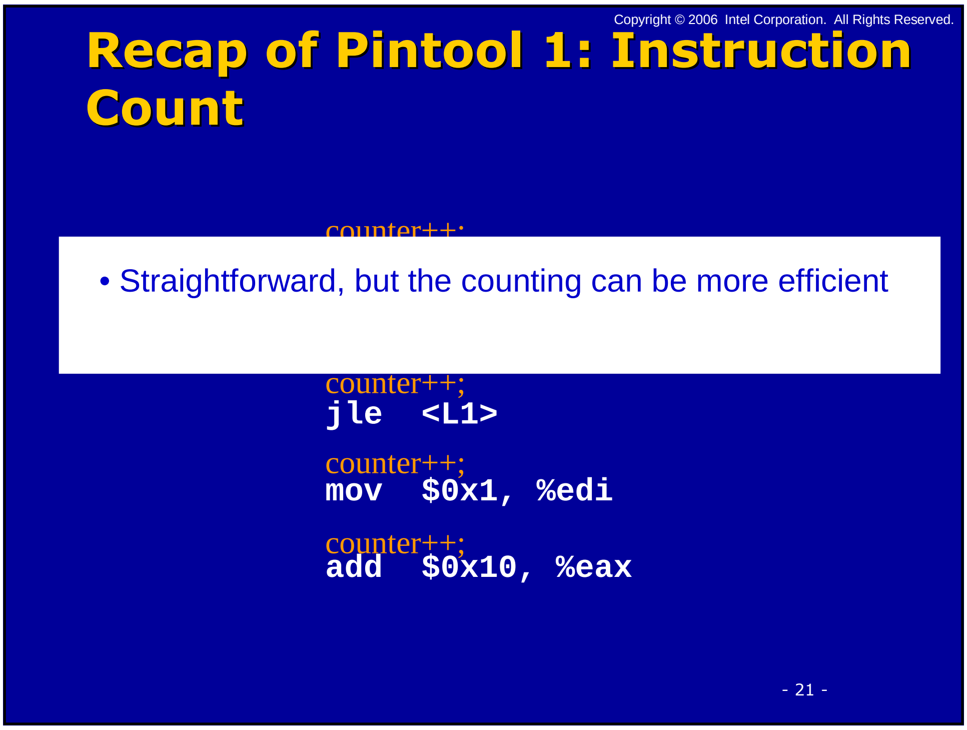


Instrumentation Granularity

- Instrumentation with Pin can be done at 3 different granular
  - Instruction
  - Basic block
    - A sequence of ins or unconditional)
    - Single entry, sing
  - Trace
    - A sequence of bas unconditional control-flow changing instruction
    - Single entry, multiple exits

1 Trace, 2 basic blocks, 6 insts

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

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

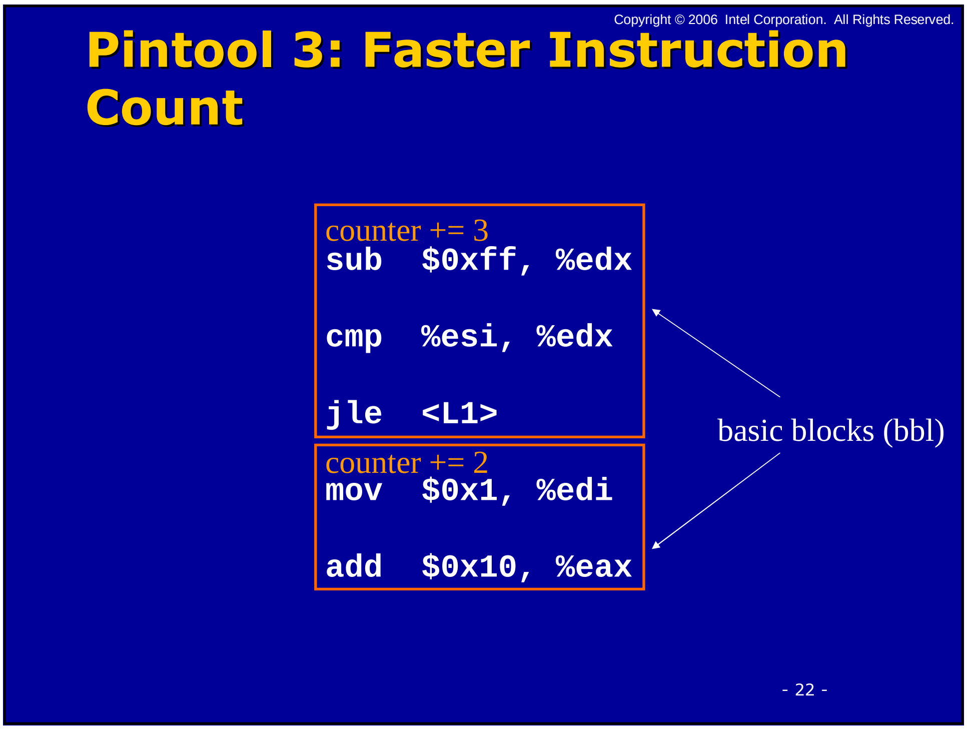


Recap of Pintool 1: Instruction Count

counter++;

- Straightforward, but the counting can be more efficient

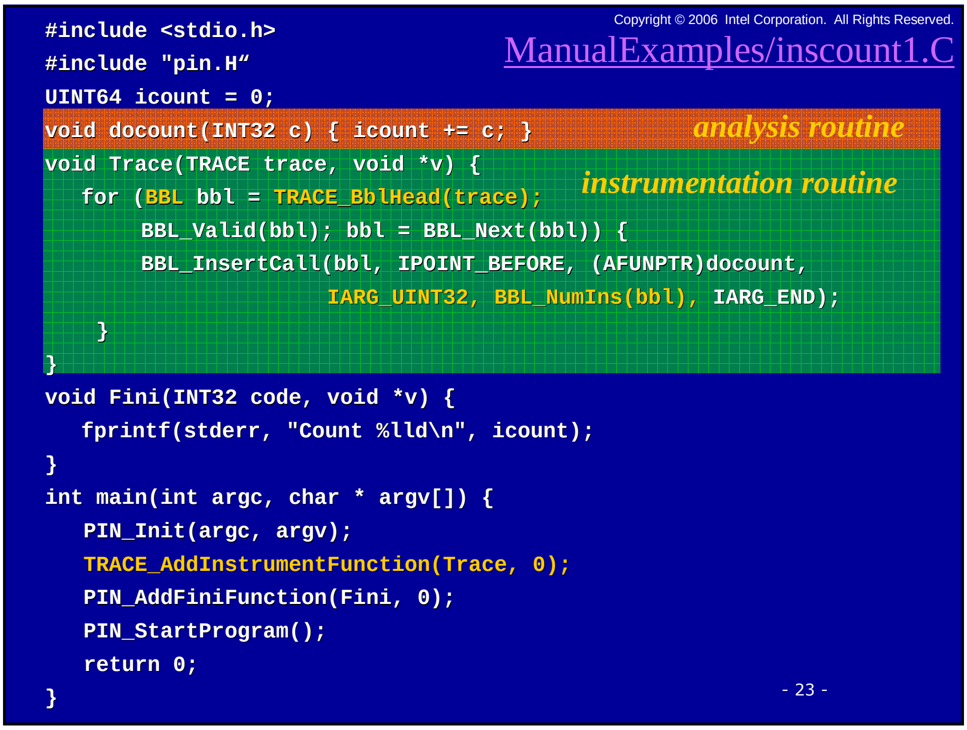
```
counter++;
jle   <L1>

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
```

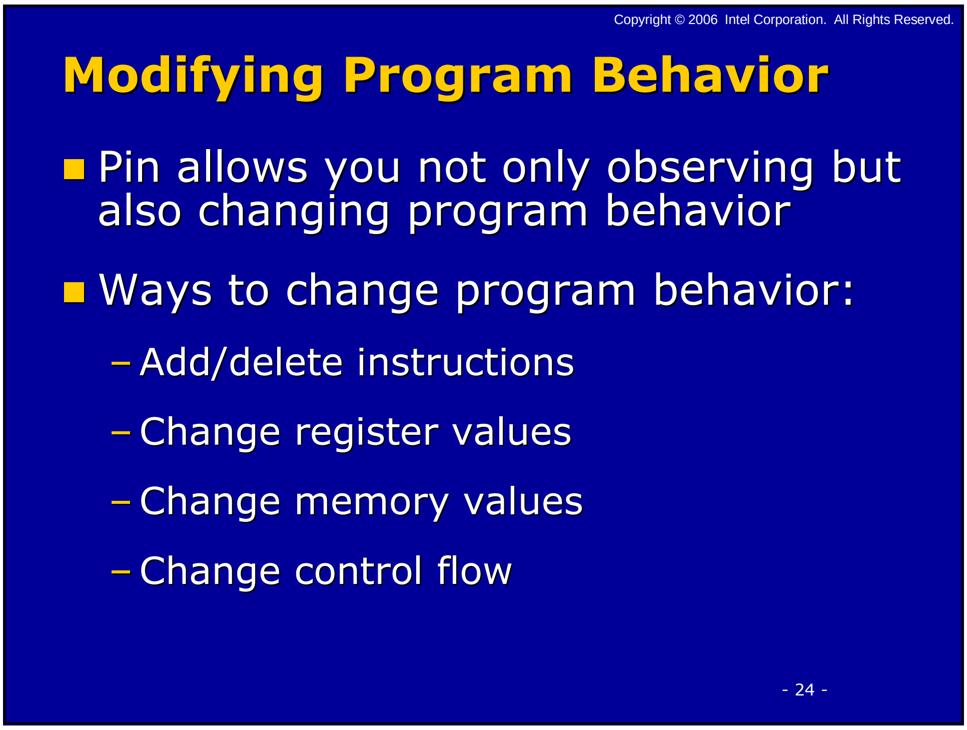


ManualExamples/inscount1.C

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

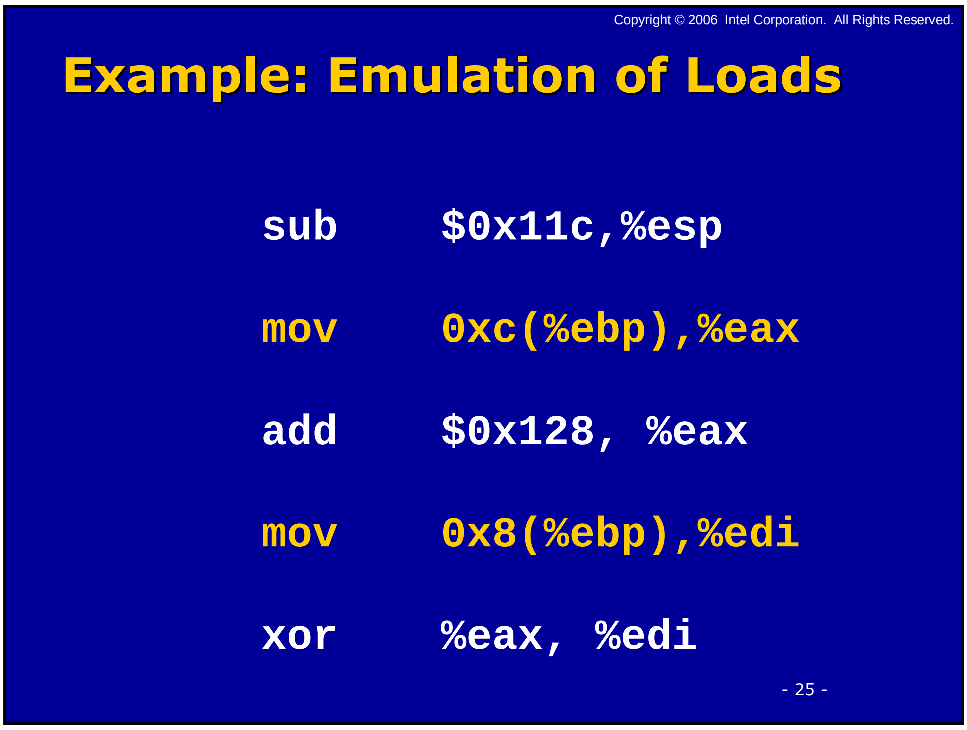
analysis routine

instrumentation routine



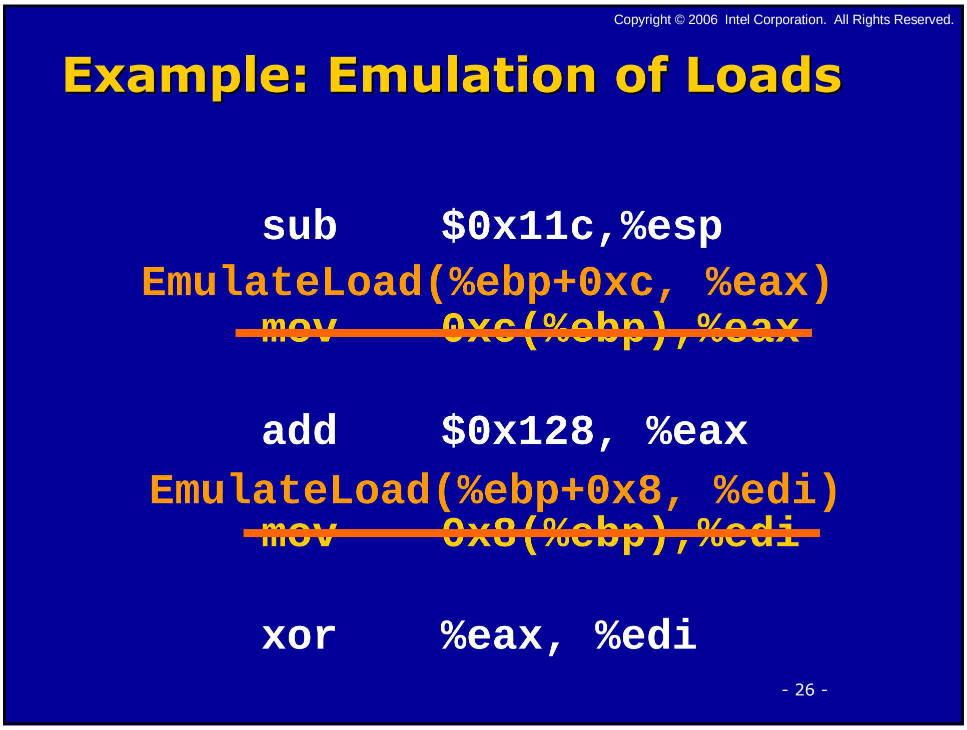
Modifying Program Behavior

- Pin allows you not only observing but also changing program behavior
- Ways to change program behavior:
  - Add/delete instructions
  - Change register values
  - Change memory values
  - Change control flow



Example: Emulation of Loads

```
sub     $0x11c,%esp

mov     0xc(%ebp),%eax

add     $0x128, %eax

mov     0x8(%ebp),%edi

xor     %eax, %edi
```



Example: Emulation of Loads

sub $0x11c,%esp

EmulateLoad(%ebp+0xc, %eax)

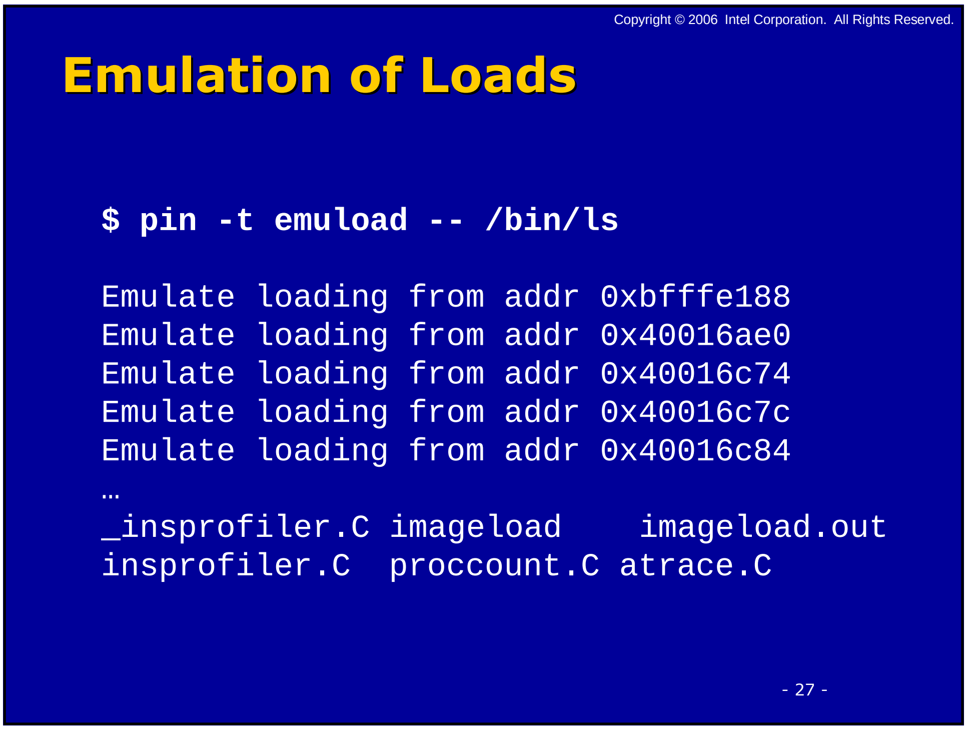
~~mov 0xc(%ebp),%eax~~

add $0x128, %eax

EmulateLoad(%ebp+0x8, %edi)

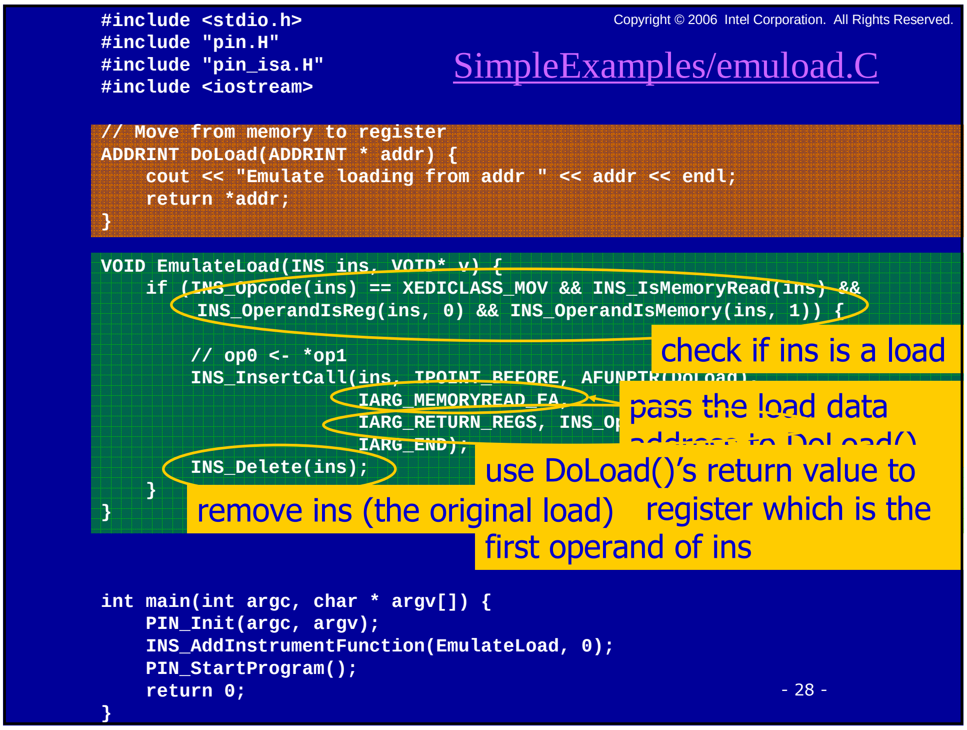
~~mov 0x8(%ebp),%edi~~

xor %eax, %edi



Emulation of Loads

```
$ pin -t emuload -- /bin/ls

Emulate loading from addr 0xbfffe188
Emulate loading from addr 0x40016ae0
Emulate loading from addr 0x40016c74
Emulate loading from addr 0x40016c7c
Emulate loading from addr 0x40016c84
…
_insprofiler.C imageload    imageload.out
insprofiler.C  proccount.C atrace.C
```



SimpleExamples/emuload.C

```
#include <stdio.h>
#include "pin.H"
#include "pin_isa.H"
#include <iostream>

// Move from memory to register
ADDRINT DoLoad(ADDRINT * addr) {
    cout << "Emulate loading from addr " << addr << endl;
    return *addr;
}

VOID EmulateLoad(INS ins, VOID* v) {
    if (INS_Opcode(ins) == XEDICLASS_MOV && INS_IsMemoryRead(ins) &&
        INS_OperandIsReg(ins, 0) && INS_OperandIsMemory(ins, 1)) {

        // op0 <- *op1
        INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(DoLoad),
                       IARG_MEMORYREAD_EA,
                       IARG_RETURN_REGS, INS_Op
                       IARG_END);
        INS_Delete(ins);
    }
}

int main(int argc, char * argv[]) {
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(EmulateLoad, 0);
    PIN_StartProgram();
    return 0;
}
```

check if ins is a load

pass the load data address to DoLoad()

use DoLoad()'s return value to register which is the first operand of ins

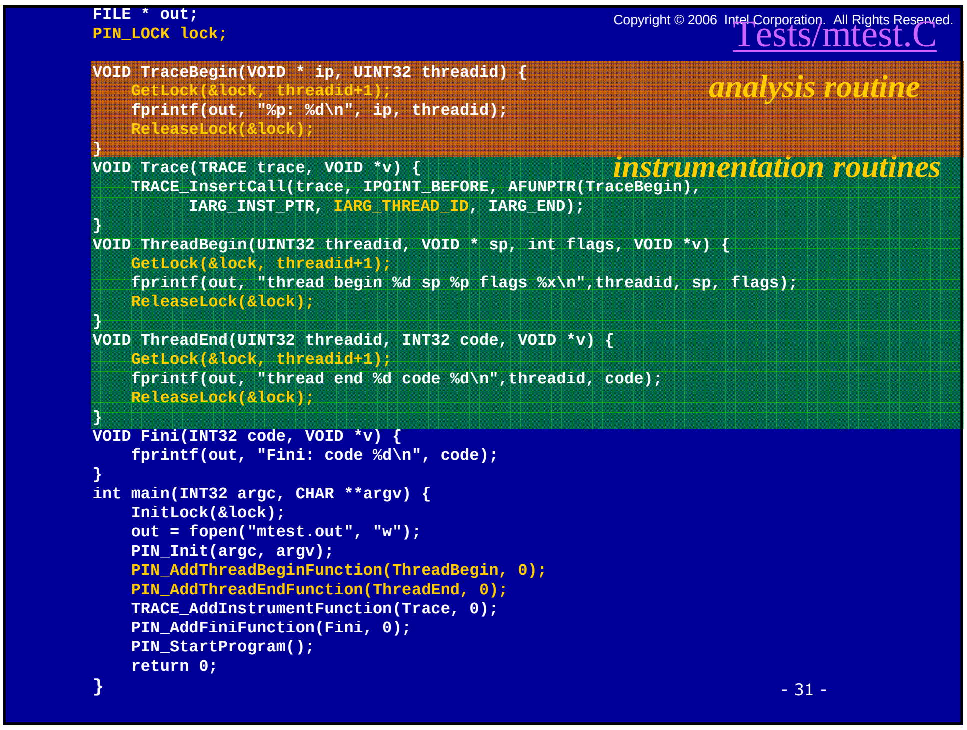
remove ins (the original load)

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

Example of Instrumenting Multithreaded Programs

```
$ 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
```

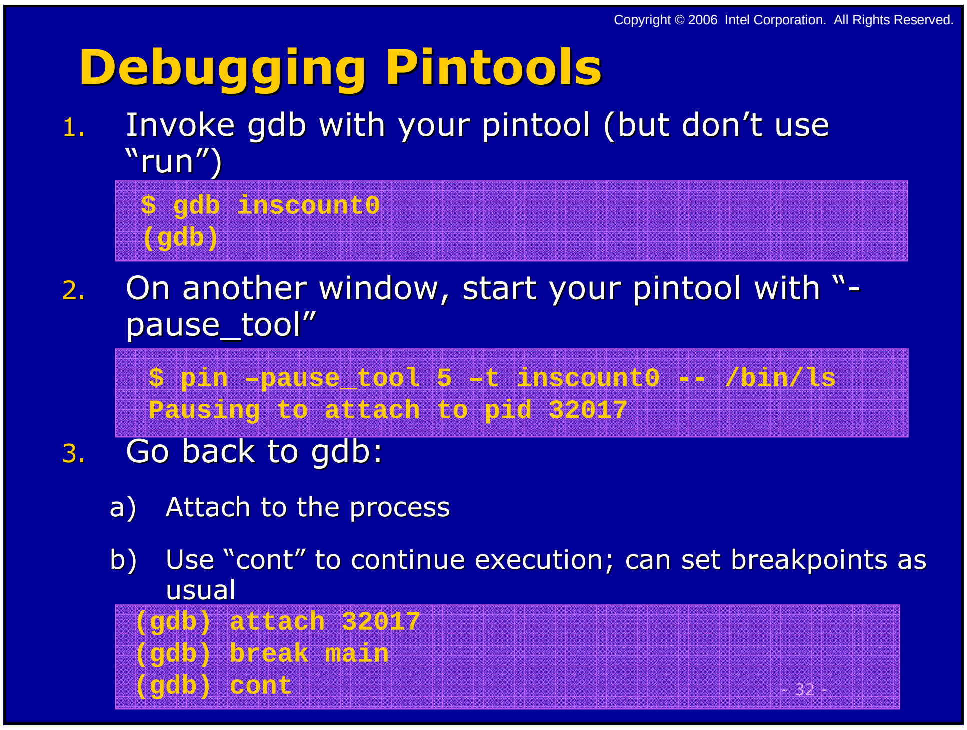


Tests/mtest.C

analysis routine

instrumentation routines

```
FILE * out;
PIN_LOCK lock;

VOID TraceBegin(VOID * ip, UINT32 threadid) {
    GetLock(&lock, threadid+1);
    fprintf(out, "%p: %d\n", ip, threadid);
    ReleaseLock(&lock);
}
VOID Trace(TRACE trace, VOID *v) {
    TRACE_InsertCall(trace, IPOINT_BEFORE, AFUNPTR(TraceBegin),
          IARG_INST_PTR, IARG_THREAD_ID, IARG_END);
}
VOID ThreadBegin(UINT32 threadid, VOID * sp, int flags, VOID *v) {
    GetLock(&lock, threadid+1);
    fprintf(out, "thread begin %d sp %p flags %x\n",threadid, sp, flags);
    ReleaseLock(&lock);
}
VOID ThreadEnd(UINT32 threadid, INT32 code, VOID *v) {
    GetLock(&lock, threadid+1);
    fprintf(out, "thread end %d code %d\n",threadid, code);
    ReleaseLock(&lock);
}
VOID Fini(INT32 code, VOID *v) {
    fprintf(out, "Fini: code %d\n", code);
}
int main(INT32 argc, CHAR **argv) {
    InitLock(&lock);
    out = fopen("mtest.out", "w");
    PIN_Init(argc, argv);
    PIN_AddThreadBeginFunction(ThreadBegin, 0);
    PIN_AddThreadEndFunction(ThreadEnd, 0);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```



Debugging Pintools

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

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

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

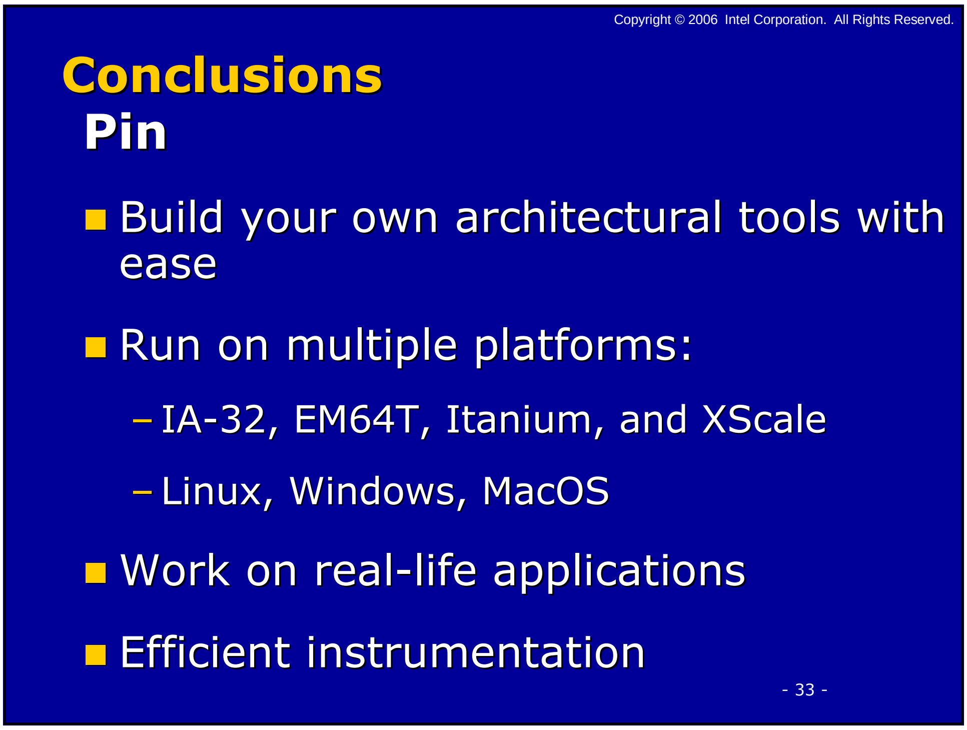
```
$ pin –pause_tool 5 –t inscount0 -- /bin/ls
Pausing to attach to pid 32017
```

3. Go back to gdb:

   a) Attach to the process

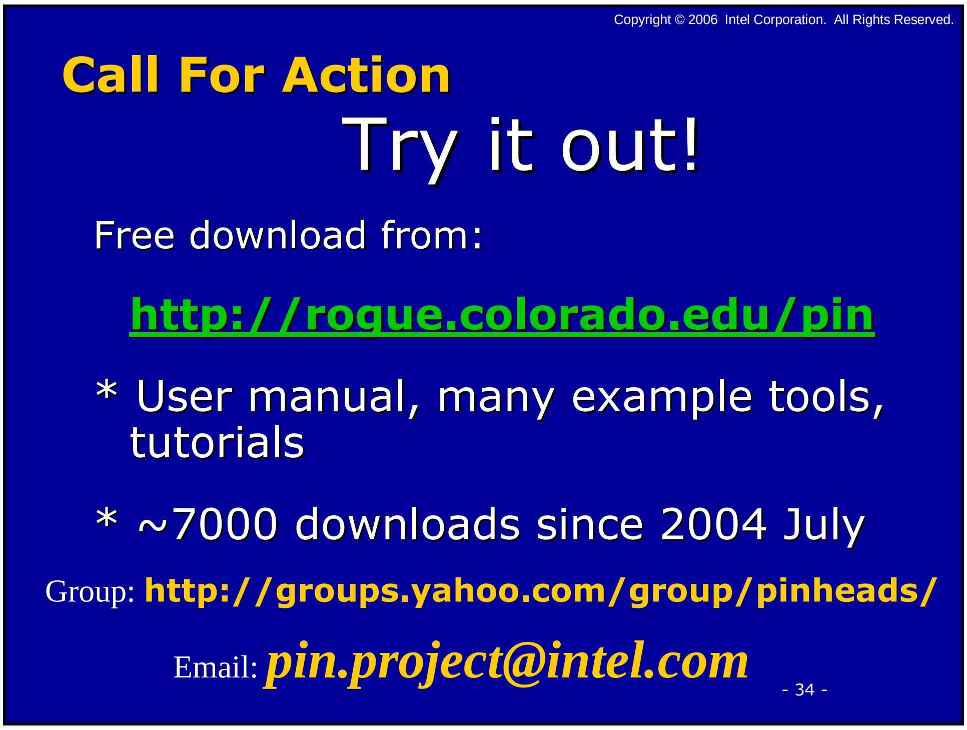
   b) Use “cont” to continue execution; can set breakpoints as usual

```
(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



Call For Action

Try it out!

Free download from:

http://rogue.colorado.edu/pin

* User manual, many example tools, tutorials

* ~7000 downloads since 2004 July

Group: **http://groups.yahoo.com/group/pinheads/**

Email: *pin.project@intel.com*