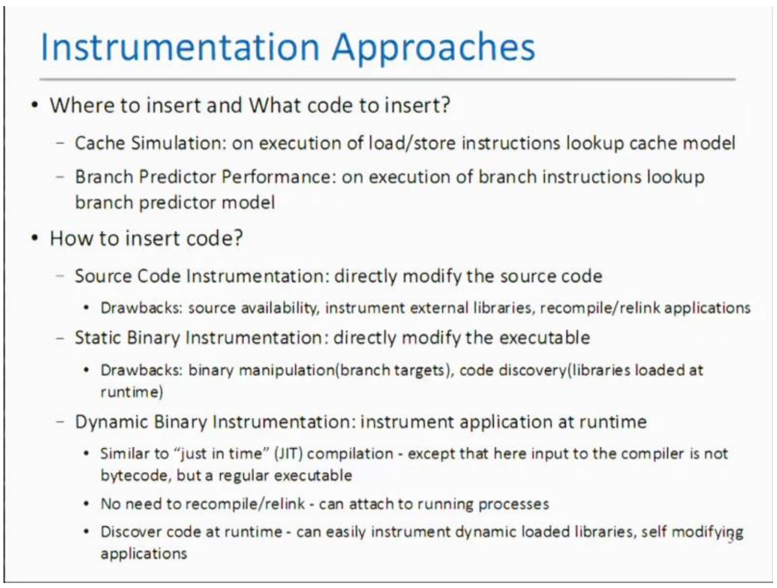
Computer Architecture

Prof. Mainak chaudhury
Indian Institute of Technology, Kanpur
Department of Computer Science and Engineering

Lecture - 09
Binary Instrumentation for Architectural Studies: PIN

We have to answer ((Refer Time: 00:14)) the where do you want to insert the code ((Refer Time: 00:18)) these are questions which are related to the problems, which are trying to solve.

(Refer Slide Time: 00:13)



Instrumentation Approaches

- Where to insert and What code to insert?
  - Cache Simulation: on execution of load/store instructions lookup cache model
  - Branch Predictor Performance: on execution of branch instructions lookup branch predictor model
- How to insert code?
  - Source Code Instrumentation: directly modify the source code
    - Drawbacks: source availability, instrument external libraries, recompile/relink applications
  - Static Binary Instrumentation: directly modify the executable
    - Drawbacks: binary manipulation(branch targets), code discovery(libraries loaded at runtime)
  - Dynamic Binary Instrumentation: instrument application at runtime
    - Similar to "just in time" (JIT) compilation - except that here input to the compiler is not bytecode, but a regular executable
    - No need to recompile/relink - can attach to running processes
    - Discover code at runtime - can easily instrument dynamic loaded libraries, self modifying applications

And the third question is actually, how do we actually do about inserting in extra code. They will be various approaches and source code instrumentation and started by instrumentation. We saw the draw backs of these. And the most popular course for instrumentation dynamic by the instrumentation. This is similar to the single digit comparison a expect that sometimes comparison, we said that we have generating code from byte code to machine code.

And dynamic by instrumentation, we have generating code from machine code to machine, while doing this generating this codes, we can the instrumentation in this insert from extra instruction extra code in the application. We are learned about ((Refer Time: 01:17)) dynamic binary instrumentation ((Refer Time: 01:19)), it provides a set of pi to 19 C or C plus plus all an instrumentation. And this instrumentation is about pen gives.

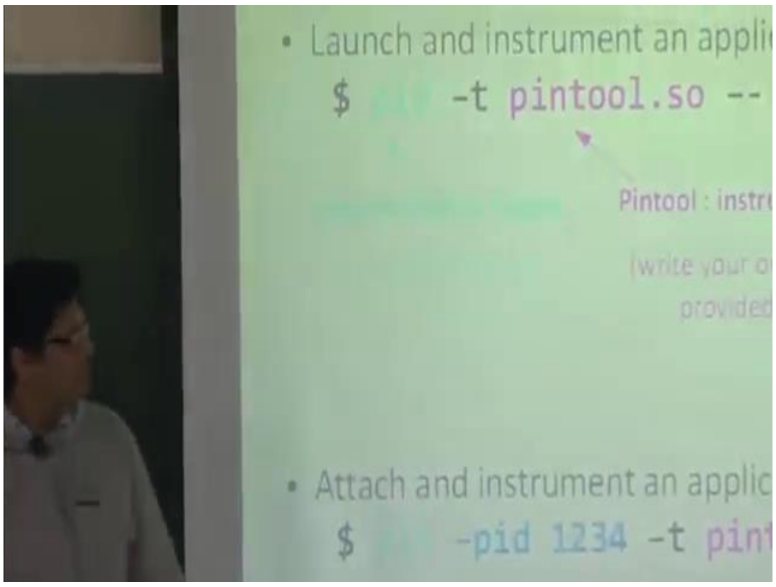
So, pen gives actually modify they are behind the thought of that means, they modify the code generations mass binary code. So, as I told you that pen tools are C or C plus, plus application, they consists of two types of routines, instrumentation routine and analysis routine. An instrumentation routine are called by pen when ever pen is about to generate code for an instruction. So, these an instrumentation routine investigate the started properties of the application.

And it decides to internet, what to this to add some extra code, but once you insert a call to analysis routine or not. And if, so it asks pens are actually inject a call to analysis routine. And analysis routine, we actually define, what the instrumentation to what to do people? So, what code, we had instrumenting that is actually the analysis routine. And important point was that, a pen tool actually register the contract with pen that what instrumentation routine to use?

So, do you understand what the call back means by a call back. So, call back is in a normal programming we actually call a functions, that function is executed at the same time will be call the functions. So, that is ((Refer Time: 03:33)) instrument execution. Then, call back, we through call back, will gives the actually register with the end time engine. That this is the function that you have to execute and it is up to the end time engine, then it is wants to execute that function.

So, just like a synchronize execution of functions. So, pin tools are registering with pen, what is the instrumentation function that will have to call. And pin will call the instrumentation function will never it is about to generate the code.

(Refer Slide Time: 04:16)



- Launch and instrument an applic
```
$ pin -t pintool.so --
```
Pintool : instr

(write your o
provided

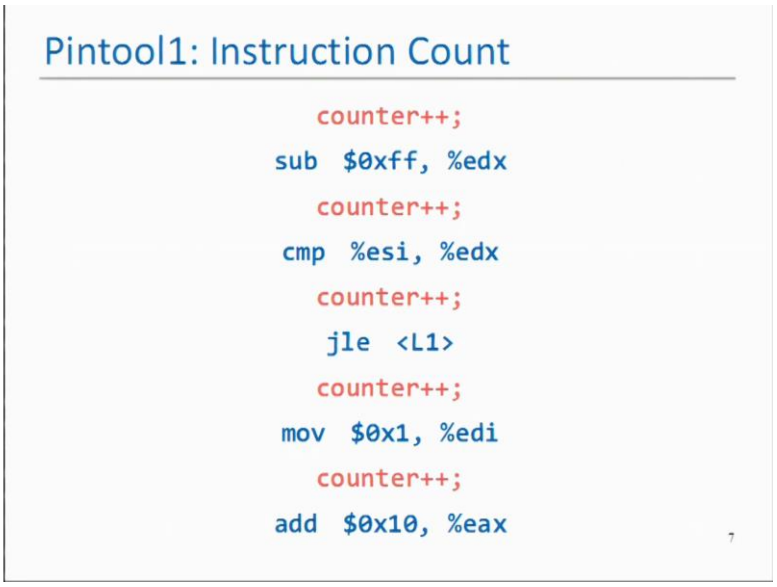
- Attach and instrument an applic
```
$ pin -pid 1234 -t pint
```

So this is, how we launch pin we give with the dash t to ((Refer Time: 04:20)) specify the pin tool and after the double dash which specify the application, that we want to instrument. We can attach pen to already an in process, through supplying the dash p i d argument to pin. Then, we saw a pen tool which was

Student: The pen actually which was compiled ((Refer Time: 04:43))

It is a dynamic library, which is build for the pen tool. So, we have to build a dynamic library. So, make for is a the pen take then there is the sample, make file provide in the printed we can directly used ((Refer Time: 05:00)). Most of the time you guys would be using modifying the pin tools, which are already de provided. So, all the build make files are already there properly.

(Refer Slide Time: 05:16)



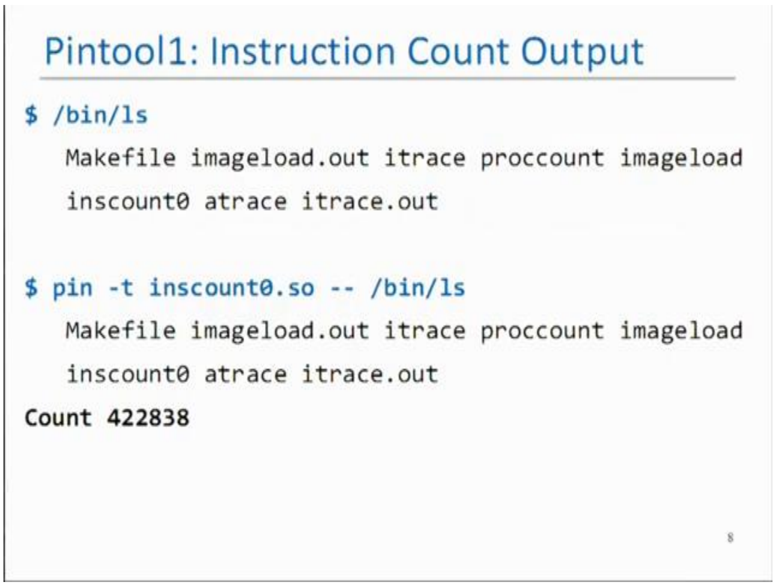
Pintool1: Instruction Count

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

7

So then, we are trying to see tool which was counting a number of instructions. They about to try to do that just before a each instruction. We would trying to insert this counter plus plus, whenever so before each instructions about that we are inserted this counter plus plus whenever this instruction is about to be executed. The counter would be incremented by 1, so this counter is actually keeping track of invalid instruction, which are executed by the application.

(Refer Slide Time: 05:46)



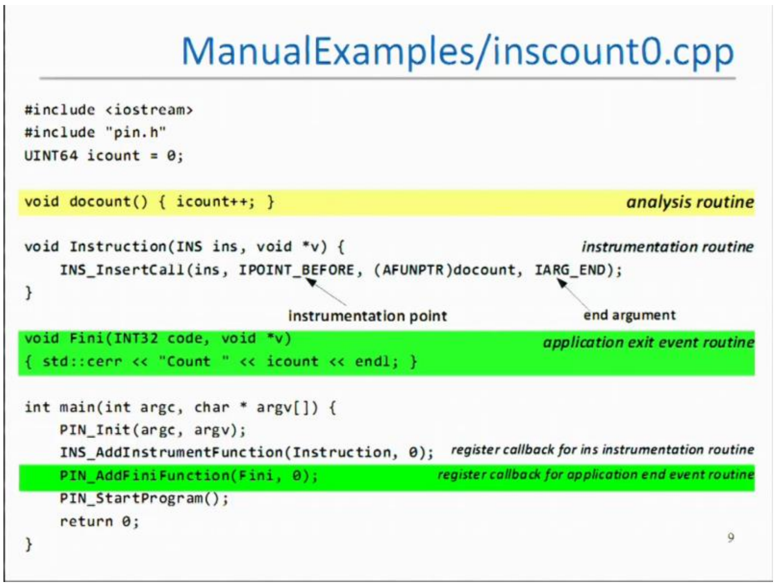
Pintool1: Instruction Count Output

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

$ pin -t inscount0.so -- /bin/ls
   Makefile imageload.out itrace proccount imageload
   inscount0 atrace itrace.out
Count 422838
```

8

So, this is how it work to, so when it run with ((Refer Time: 05:51)) displays the number of instruction that are executed, we are give an instrumenting the ((Refer Time: 05:56)) application.

(Refer Slide Time: 05:58)



ManualExamples/inscount0.cpp

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

void docount() { icount++; }                                    analysis routine

void Instruction(INS ins, void *v) {                        instrumentation routine
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}
                              instrumentation point                 end argument
void Fini(INT32 code, void *v)                          application exit event routine
{ std::cerr << "Count " << icount << endl; }

int main(int argc, char * argv[]) {
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);  register callback for ins instrumentation routine
    PIN_AddFiniFunction(Fini, 0);                register callback for application end event routine
    PIN_StartProgram();
    return 0;
}
```

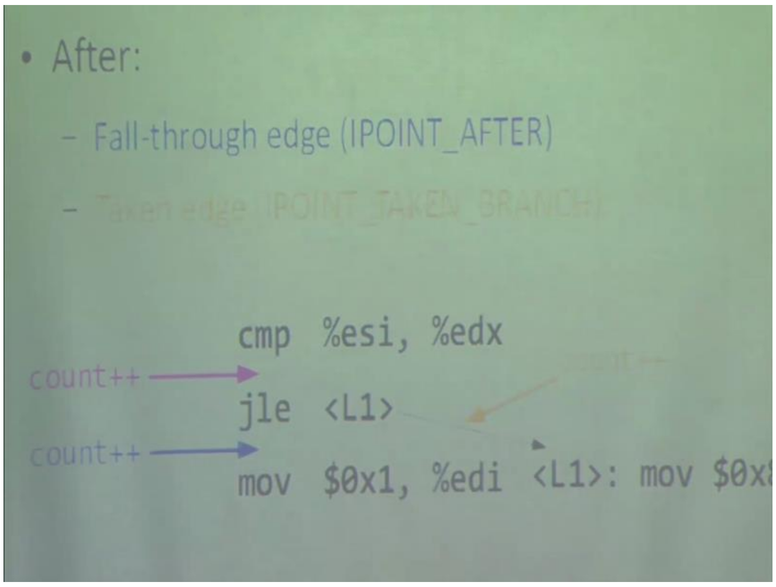
9

So that, this is the code for the INS count, it act so do count is the analysis routine, instruction was the instrumentation routine. And here INS add instrument functions, he will be registering, the call back for instrumentation routine with pin. So, it is saying that whenever pin is about to generate code for each instruction, it should called the analysis routine for it.

And in analysis routine, we were saying that insert a call to this do count function, analysis function. Just before the code for this instruction. Insert a call to the do count analysis function, and the do count just including the countable and uncountable. And then this, you registering a call back for an application end event. This whenever the application is about to end the Fini function would be called a Fini function just prints the I count value.

So, focus on this instrumentation call INS insert call. So, the first argument to INS insert call functions actually to this function call a pin tooling asking pin tool injector call to analysis routine. So, this function takes an input the instruction for which instruction, it is asking to inject call the position. This position is instrumentation related to the instruction. To see board and then the analysis routine and this is the place marker for the end of arguments to insert call.

(Refer Slide Time: 07:54)



- After:
  - Fall-through edge (IPOINT_AFTER)
  - Taken edge (IPOINT_TAKEN_BRANCH)

```
count++ →
count++ →
```

```
cmp  %esi, %edx
jle  <L1>
mov  $0x1, %edi   <L1>: mov $0x
```

Analysis routine actually specify the position related to an instruction, where pen should inject this presentation called is a position related to an instruction, where pin should inject a call to the analysis routine. So, this j l e instruction is the branch instruction, so the conditional a branch instruction. Here, it is doing the comparison when from flats would be insert and j l e would inspect those flats in based on the x will either travels this path or to jump to this.

So, then the instrumentation point is I point. So, the analysis called is injective just before this instruction. And then, the instrumentation point is the fall through I point after, then the analysis called injected on the fall through edge. By fall through edge mean that the instruction, which is next to that instructions in the source program. So here, the MOV instruction is next to this j l e instruction in the source program.

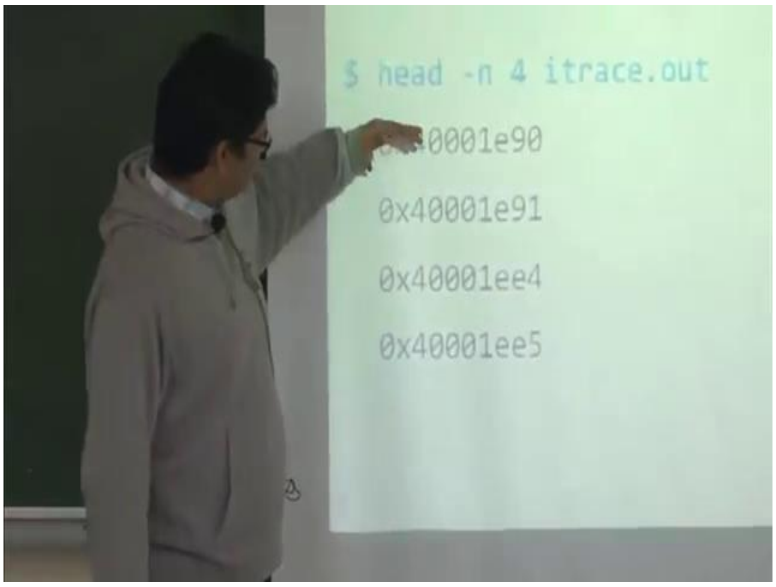
So, this definition of fall through edge actually, so fall through edges may not always exist. Suppose, that here it was a conditional branch, so they were two parts. But, it is this was an unconditional jump instruction, there would a single part which pin for source, only a single control floor path. So, in suppose this was a conditional jump, then this path should always be taken. Unconditional jump in this path should always be taken.

So, the fall through edge would not always exist. And, if the instrumentation point was I point taken back, then it actually injects the call on the branch taking path, you clear about this?

Student: So, this does not exist, then it is prevent to calls it is psi after.

It, so suppose fall through does not exist, then instead the path psi after the code actually insert the analysis, you called that path does not exists. Then, the ((Refer Time: 10:36)) analysis called the path that not exists. Then, these are the second instrumentation to trying to pin tool instruction.

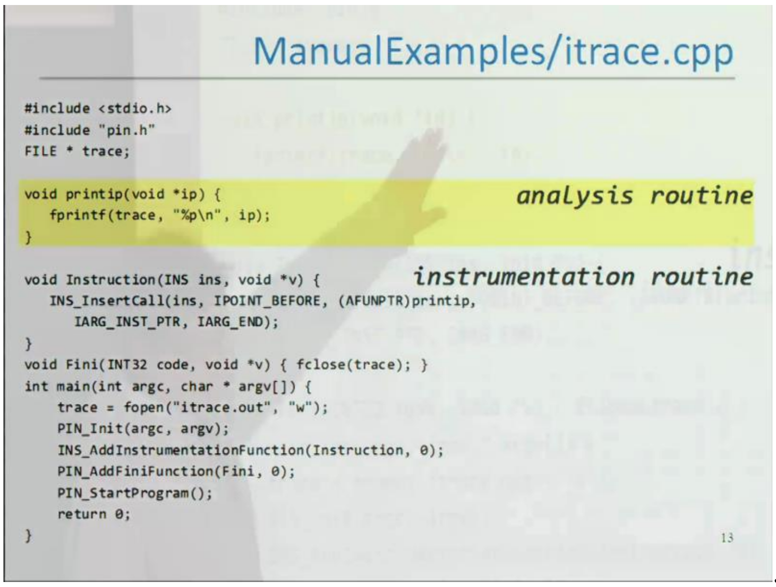
(Refer Slide Time: 10:51)



```
$ head -n 4 itrace.out
0x40001e90
0x40001e91
0x40001ee4
0x40001ee5
```

So, that instrumentation causes just a list of instruct them, so these are the instruction addresses, which are executed. We are actually trying to do is that ((Refer Time: 11:02)) just before instruction. We go trying to insert the call to this pin tip function, which takes the input instruction address and just print. So, there will be passing arguments to the analysis function.

(Refer Slide Time: 11:27)



ManualExamples/itrace.cpp

```
#include <stdio.h>
#include "pin.h"
FILE * trace;

void printip(void *ip) {                          analysis routine
   fprintf(trace, "%p\n", ip);
}

void Instruction(INS ins, void *v) {       instrumentation routine
   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_AddInstrumentationFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

13

.

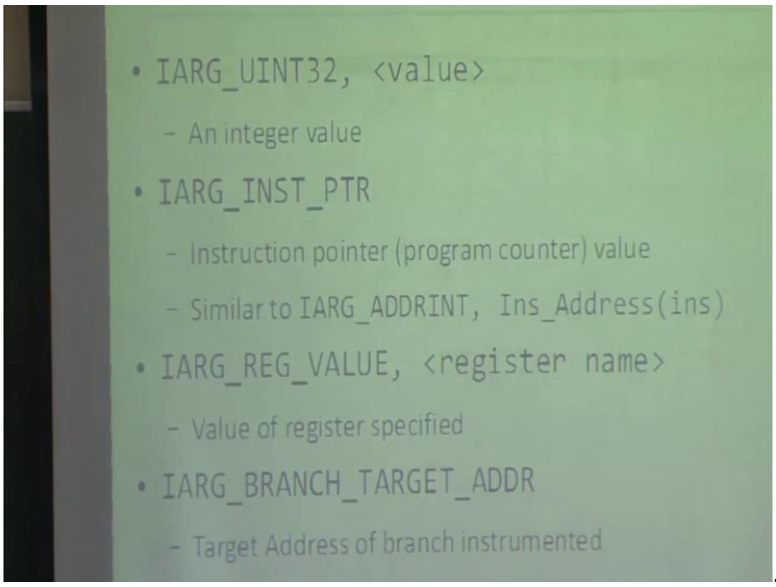
So, ((Refer Time: 11:26)) this similar to the last pin tool. So, let us look at the analysis routine. In the analysis routine, you will be takes us input the instruction address and it just prints in the trace file of the instruction pointer. In the instrumentation routine, the ((Refer Time: 11:46)) same except that the input arguments to this are no change. We are passing an additional IARG instruction pointer argument.

This IARG instruction in pointer argument actually so, it is like a macro it asks them, so whenever this is past the instruction. The address of the current instruction, which we are trying to instrument that address would be passed the input argument to the analysis routine. So, whatever it is plays after the analysis function, in this function. And before the IARG end good we passed at the input to the analysis routine. Here, IARG and IARG instruction pointer is passing through.

Student: So, we need not put pass through IP, we need not tell the group of them.

It will automatically, so here it know what instruction it is instrumentation. So, it can determined on it is on, what is the instruction are those.

(Refer Slide Time: 13:10)



- IARG_UINT32, <value>
  - An integer value
- IARG_INST_PTR
  - Instruction pointer (program counter) value
  - Similar to IARG_ADDRINT, Ins_Address(ins)
- IARG_REG_VALUE, <register name>
  - Value of register specified
- IARG_BRANCH_TARGET_ADDR
  - Target Address of branch instrumented

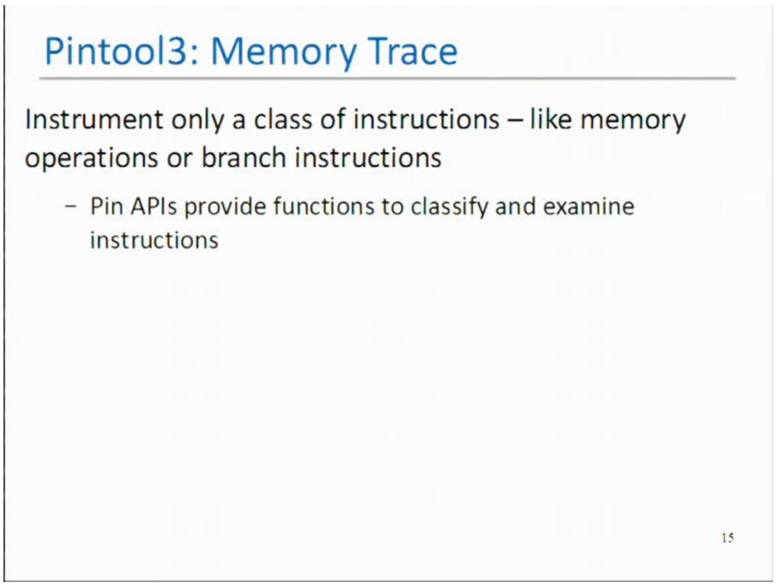
.

They are also the un thumb, so they are lot of a more argument, which we can so suppose at we what to pass 32 bit integer value. So in the previous example, you just passing a single IARG instruction point, most of time this is not we pass two arguments. The first one specifies the type of the argument that we are trying to pass to the answer and the second was the actual balance. So suppose at, we want to pass 32 bit integer value to the analysis routine.

So in the arguments, which specify IARG 32 and then in the second argument would be the value that we want to pass? So, this is kind of a short hand that in provide. So, here we do not have to express these specify the value. We can do something like this we can say that IARG address integer, so this is a type for in addresses. So, we are trying to pass an address after their instructions and then this is the actual value address instruction. So, this IARG instruction point is the short hand for this.

We can pass these two values, so there is the type of register value and this is the register name code two value old bar. Similarly, this is also shorthanded directly determines from the current instruction information, what is the branch target the address. An IARG MEMORYOP EA, here we specifying an instruction can have from number of MEMORYOP EA, memOp is the index of the memory operands and this is the type of the MEMORY EA. If the effective address of the memory operand, the many more input argument then it should beat the manuals are...

(Refer Slide Time: 15:11)



Pintool3: Memory Trace

Instrument only a class of instructions – like memory operations or branch instructions

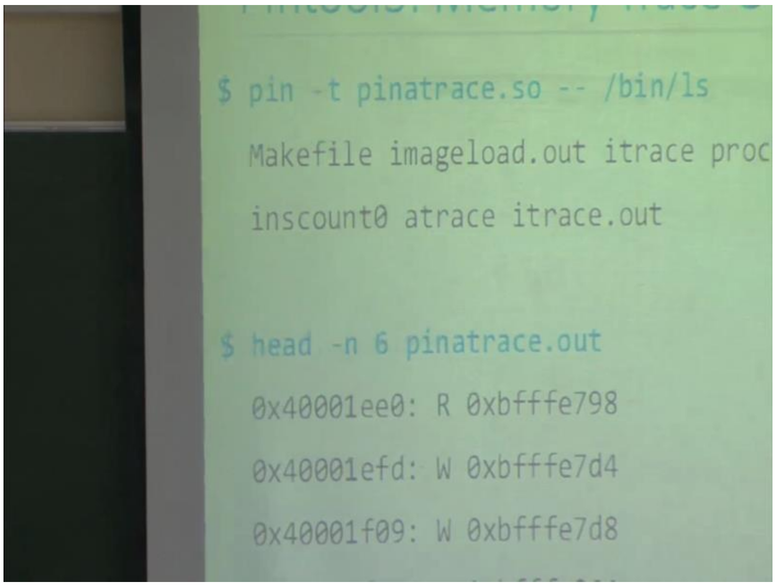
- Pin APIs provide functions to classify and examine instructions

15

Now, it is coming to what we have doing in that, we have been instrumenting each instruction in the application. So, we register, so we have been instrumenting rejecting a call to the answers from should just before each instruction. So, most of the time, that is not necessary, we would actually want to instrumental specific class of restrictions. Suppose that we want to do memory, suppose that we want to do cash stimulation.

So here, we are only interested in instrumenting instruction, which perform which do a load or restore. So, we are actually interested no need instrumenting the judges. So, pin provides APIs functions through which we can examine and classify whether, what type of instruction this is. So, we would use the pin API functions to examine whether an instructions a load is doing load only if it is doing a load or restore, we would inject a call to the analysis routine.

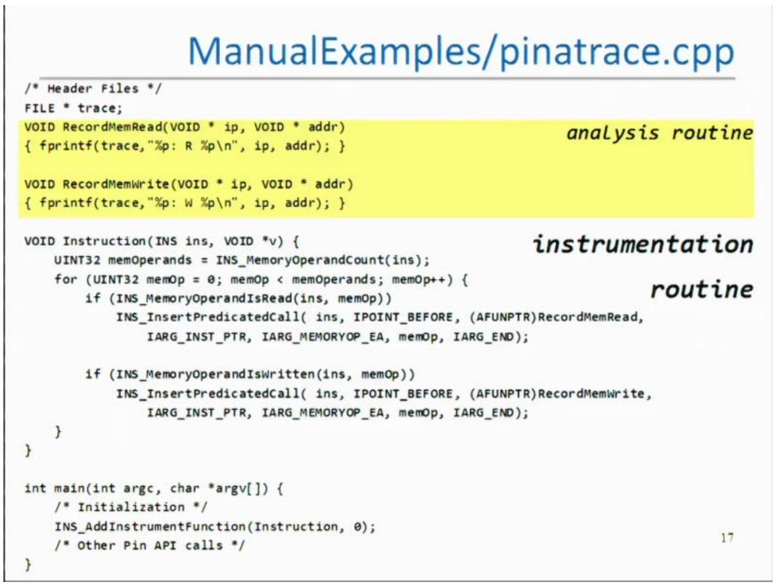
(Refer Slide Time: 16:33)



```
$ pin -t pinatrace.so -- /bin/ls
  Makefile imageload.out itrace proc
  inscount0 atrace itrace.out

$ head -n 6 pinatrace.out
  0x40001ee0: R 0xbfffe798
  0x40001efd: W 0xbfffe7d4
  0x40001f09: W 0xbfffe7d8
```

So, this is a pin tool for printing of the memory trace. The memory trace looks like this, which line near is the instruction address which did this load pin tool store. And then ((Refer Time: 16:44)) either it whether it was a load whether it was a read or a write. And then the memory address on which it was doing the load or restore. So, it cannot taken trace got this was the tool it have been taken from so it understood.

(Refer Slide Time: 17:04)



ManualExamples/pinatrace.cpp

```
/* Header Files */
FILE * trace;
VOID RecordMemRead(VOID * ip, VOID * addr)
{ fprintf(trace,"%p: R %p\n", ip, addr); }

VOID RecordMemWrite(VOID * ip, VOID * addr)
{ fprintf(trace,"%p: W %p\n", ip, addr); }

VOID Instruction(INS ins, VOID *v) {
    UINT32 memOperands = INS_MemoryOperandCount(ins);
    for (UINT32 memOp = 0; memOp < memOperands; memOp++) {
        if (INS_MemoryOperandIsRead(ins, memOp))
            INS_InsertPredicatedCall( ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
                IARG_INST_PTR, IARG_MEMORYOP_EA, memOp, IARG_END);

        if (INS_MemoryOperandIsWritten(ins, memOp))
            INS_InsertPredicatedCall( ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
                IARG_INST_PTR, IARG_MEMORYOP_EA, memOp, IARG_END);
    }
}

int main(int argc, char *argv[]) {
    /* Initialization */
    INS_AddInstrumentFunction(Instruction, 0);
    /* Other Pin API calls */
}
```

Now, just look at the pin tool, so this is the simple see applications. So, these are the analysis routines. So, record amen read, so this will be called whenever ((Refer Time: 17:17)) so a memory read. And it is printing the first argument is taking the instruction point out if you saw in the pin tool. And then, it is taking the address this is the memory operand address. The address of the memory operand, it is printing the instruction point address the read and then the memory operand.

Similarly, features record ((Refer Time: 17:44)), I am not shown then ((Refer Time: 17:50)) report. So, I am only doing instruction instrumentation and the instrumentation function is instruction. So, if you look in the instruction function, so this function would be called whenever we had trying to generate code for end for an instruction. So, this is a pin ((Refer Time: 18:18)) function through which we get the number of memory operand this. So, the necessity instruction can perform more than then load store.

Here, I am counting the number of memory operands in a instruction. So, ((Refer Time: 18:34)) instruction the more than one memory operand. And then, I am iterating over the memory operand. And I am finding out whether it is doing a memory read or a memory write. So, ((Refer Time: 18:47)) memory operands can be both the source and the destinations, same operand can be use both are the source estimated. So, I check for both whether, it is there is no else both are individual with it.

So, if it is doing a memory read, then I insert a call, so this forget about this INS predicated call. INS insert predicated this is almost similar to INS insert path which we saw previously. And we pass the INS the instruction, the position instrumentation point INS before, then because this is the read so we call the read analysis routine. And then, we pass the instruction address just like in the previous. And we have to also pass the memory operand.

So, the memory operand is pass like this, so we IARG memory operand EA and then we give the memOP. So, we discuss this previous this is what a short hand and here we are passing through a arguments, so the both could be use to pass the second argument. You get, what was happening here? And similarly for right, whenever it is a right we just changing a calling a memory right an analysis function.

So here, we are injecting, so suppose that the instruction is doing both read and write they say instruction, so we are injecting first to analysis calls to two different analysis routines. And, whenever that is about that, if the analysis routine is about to be executed it will pass the instruction pointer. And it will pass the memory operand. So now, what happens is that the memory operand in most of the instruction, determined at the grunt time, when in the instruction is actually going to be executed.

So, when they are doing the instrumentation at that point we do not know what is the memory operands address, that could only be known when the instruction is about to be executed. So, first question was actually that, why can we pass see the so when the instruction is about to be executed at that point been has to again look up the instructions, environment you find out the memory operand.

Student: ((Refer Time: 21:16))

It get so this MEMORYOP_EA and memOP this would get converted into some, if you find out the instructions memory operands address. And that would be pass to the record memOP.

Student: Here, I gives and I could also contain terminates the kinds of read and write. But here, doing it provides the same address ((Refer Time: 21:47)).

So, we know that information that before at just like decoding the instruction.

Student: so that I understand the point out ((Refer Time: 21:54))

((Refer Time: 22:07)) so I am not displayed that memory operand. So, we can also pass the programs contacts, the execution contact as the input argument to an analysis continue. So, we can,

Student: Admission at our.

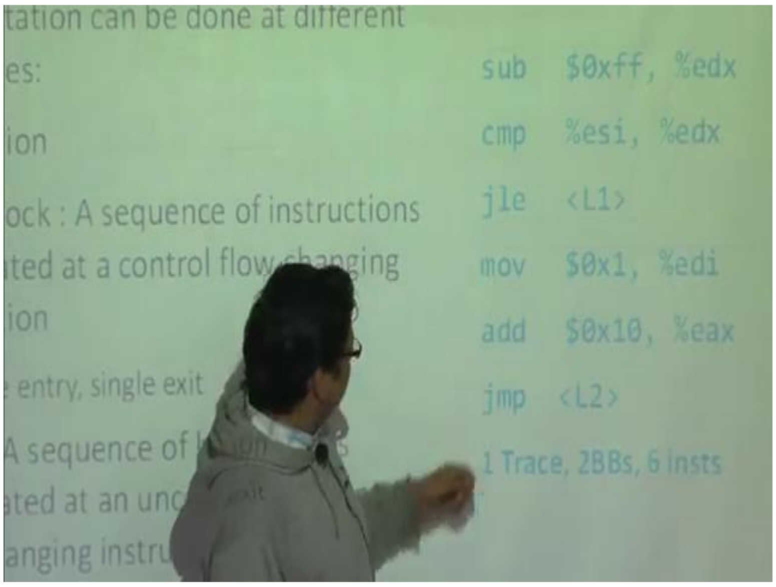
But, the over head of that is petal the pin instrumentation is actually slow ((Refer Time: 22:34)). So, you we should most of the time do most of the work were in the instrumentation time, because that is called once and the analysis routine should act to the minimum work. So, are you clear about this?

Student: So, ((Refer Time: 22:51)) predicate count.

I forgot, so here we are doing INS insert predicated call I mean, so ((Refer Time: 23:03)) call some instructions, which have some predicate before and only the predicate is true, the instruction is actually executed. So, the predicated itself a for part of the instruction only the predicate is true the instruction would actually be executed. So, this predicates are like some conditions are, so talk about grand value. so till now we happening a registering a call back to an instruction instrumentation routine.

So, this function actually said when ever pin was about to generate code for an instruction. It should call the instrumentation routine. So, we would actually instrumenting at the graduality of each instruction. So, you would calling the instrumentation routine for each instruction. The over head of this is quiet last, so pin provides as the instructions graduality.

(Refer Slide Time: 24:16)


tation can be done at different
es:
ion
ock : A sequence of instructions
ted at a control flow changing
ion
entry, single exit
A sequence of
ated at an unc
anging instru

```
sub  $0xff, %edx
cmp  %esi, %edx
jle  <L1>
mov  $0x1, %edi
add  $0x10, %eax
jmp  <L2>
```

1 Trace, 2BBs, 6 insts

You can call the analysis instrumentation routine add basic block. So now, it so see what these are, so basic block is the sequence of instruction, which are terminated by control flow changing instruction. So, whenever you have a suppose this is the pins code sequence. And so this was the control flow instructions. So, this is one basic block and starts this is second basic block.

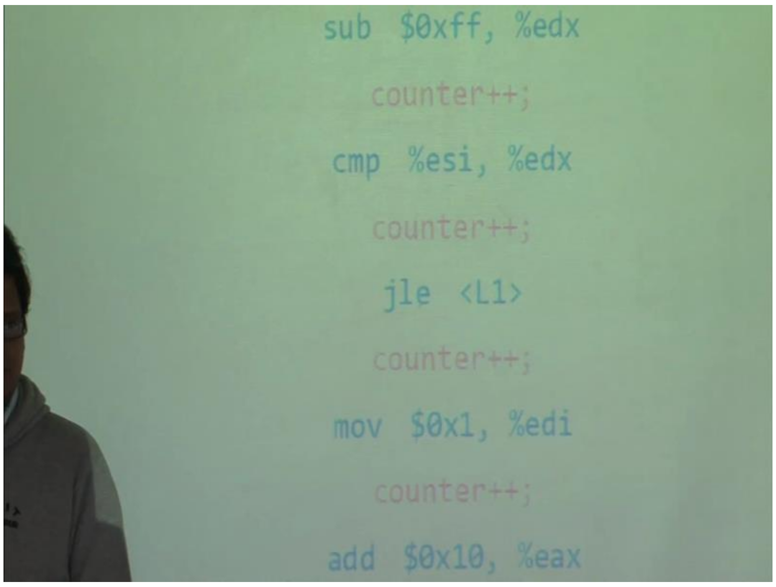
And then, there is trace ((Refer Time: 24:49)) the sequence of basic block, which are terminated at the unconditional control flow changing. So hold the single trace, so the trace want to terminated, because the tray system unconditional flow. So pin provides you with these other EA calls to register instrumentation routines for doing a basic block level instrumentation and trace base instrumentation. So, when ever say trace base instrumentation that means, when ever pin is about to generate code for a trace.

It should call the analysis routine. Now, we were saying when ever pin is about to generate code for the instruction it should call the analysis routine. So, we have seen instruction base instrumentation we would see the example of these and then the fourth type is routine based instrumentation. Routines are function, so this is almost instruction it says that when ever pin is about to generate code for instruct for a function, it should call the analysis routine act.

Student: Apart from the instruction ((Refer Time: 26:12)) the reciprocal trace given ((Refer Time: 26:17)).

We will see ((Refer Time: 26:18)).

(Refer Slide Time: 26:24)



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

So, this was the instructing count, so we were injecting this counter plus, plus that we for this instruction. So, this is not that mechanism of counting instructions. Because, we have to there lot of addition that this code would have know to that. So, this is the straight forward mechanism, but it is not that efficient. Can you kind of another type of what how can you optimize for that.

Student: ((Refer Time: 26:56))

To count the number of instructions, but can we do some out, I mean reduce this number of analysis cause that we are inserting. So, this is the same courts it consists of two basic blocks. And basic block as a single entry point and a single exit point. It does not, so what we can do is that, just before the start up the reciprocal we can count the number of instruction in the basic block. And we can just including the counter by there.

So, this would be this, so instead of doing three additions we are doing single addition, similarly for the second basic block.

Student: So, we also of course the points are basic block statistic ties, count the number of instruction in the basic block is true ((Refer Time: 28:00)).

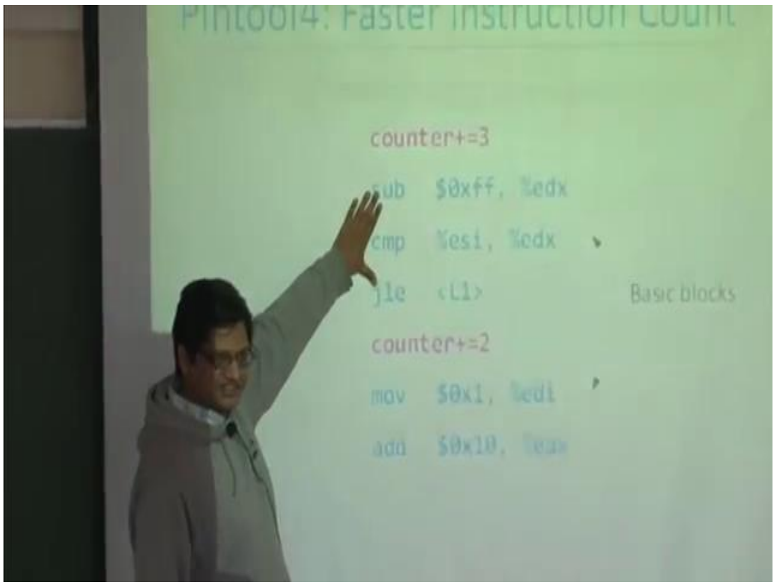
Because, so you would see...

Student: how we get the counter. Now, otherwise infinite counting, that means, how will be grow free ((Refer Time: 28:18)) here seeing, the counting number of instruction are executed and contain the inputs; that means doing same thing. Only that describes the three instructions are estimated and it ((Refer Time: 28:33)).

That because the basic block at the single entry from single exit. That is why, because of that the number of instruction would remain the same.

Student: ((Refer Time: 28:54))

(Refer Slide Time: 28:58)

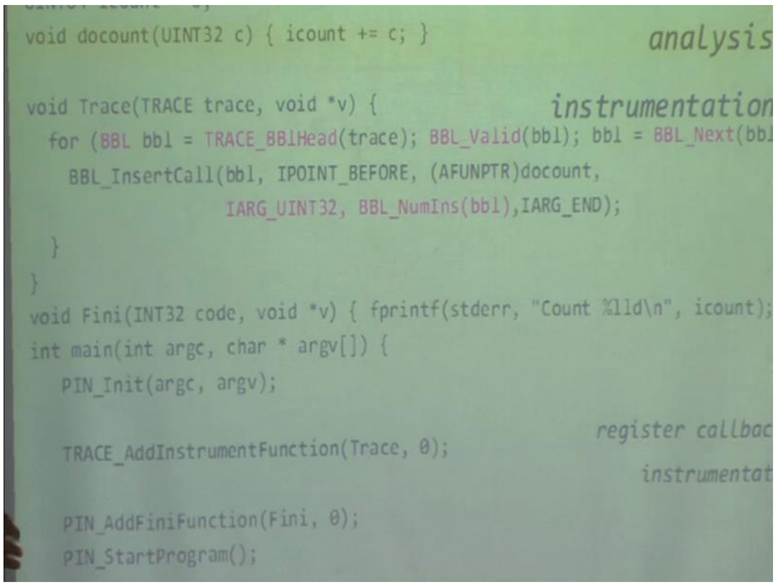


Pintool4: Faster Instruction Count

```
counter+=3
sub   $0xff, %edx
cmp   %esi, %edx
jle   <L1>
counter+=2
mov   $0x1, %edi
add   $0x10, %eax
```

So, we have to figure out, what are the basic blocks and we have to count the number of instructions in the basic block and the instructions. So, bit sees the code for this.

(Refer Slide Time: 29:09)



```
void docount(UINT32 c) { icount += c; }                    analysis

void Trace(TRACE trace, void *v) {                  instrumentation
  for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bb
    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);
                                                      register callbac
   TRACE_AddInstrumentFunction(Trace, 0);
                                                         instrumentat

   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();
```

So, this is the awareness routines has change it has a input argument see, which it increment the I count by that see. And this is our instrumentation routine we will come to this. Let us, focus on this like a call back function, so instead of doing INS add instrument function we are doing trace add instrumentation function. So here, we are doing registering a call back for trace instrumentations. So, whenever pin is about to generate code for a trace, it will call this a trace instrumentation function.

Now, it is look at the instrumentation function here, what we are do, so a trace for just a sequence of basic block. So, it can be thought of for the list of basic blocks. So here, in the instrumentation function we are actually, I grating over the basic blocks in the trace. So, we get the head, so trace is the bits of basic block. So, we get the head pointer and we are changing whether the basic blocks is this is 1 0 level and then this is the end.

And, each I ((Refer Time: 30:22)) of this formed we are saying, we asking to inject a call at the starts of the just, before you generate the code for the basic block. So, case you got doing INS insert call that was saying just before it generates code for the instruction it should inject call to the analysis. You have to gave same that just before it generates code for the reciprocal we should insert a call to the analysis ((Refer Time: 30:46)).

So, we are I granting over the basic block. And we have this we inject a call, so for each basic block we inject call to this ((Refer Time: 31:00)) function. The input arguments here are see, we pass 32 IARG 32 and this is the value that we writing past. We can find out the, so this the BBL function, which guess us the number of instructions in the,

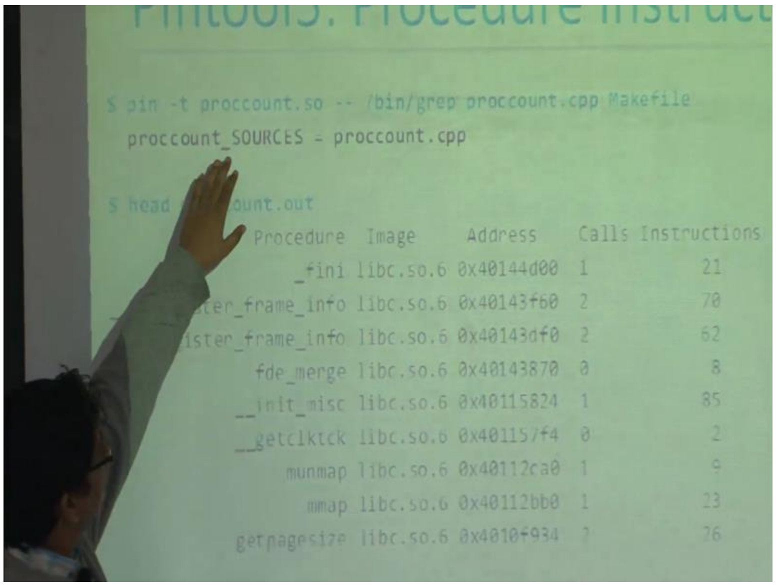
Student: Can dissolves execute the instrumentation address can write ((Refer Time: 31:19))

We have this information act instrumentation will how many instructions are there in the ((Refer Time: 31:40)). And we inject call to this ((Refer Time: 31:44)) function with the number of instruction in a basic block at the start of the basic block. So, here the instrumentation function is taking as input the trace for which pin is about to generate code. And the second argument is the second argument path going we do not used, so just pass 0.

So, do you look at this, here I said the reason do instrumentation at different Grammatik. We can do instrumentation at basic block. So, you actually are doing instruction, in the example actually calling the instrumentation function at the Grammatik of trace. And we were injecting analysis routine at the Grammatik of basic blocks. There is no specific instrumentation routine to register call backs of basic block. So, the only two instrumentation resist the two one add instrument function.

One INS adds instrumentation and there is a trace, add you can only register a instrumentation call back for other instruction or for the trace level. And then, the trace level we can I trace over the basic block. So, there was one more the routine, now we see example for how do this routine based instrumentation.

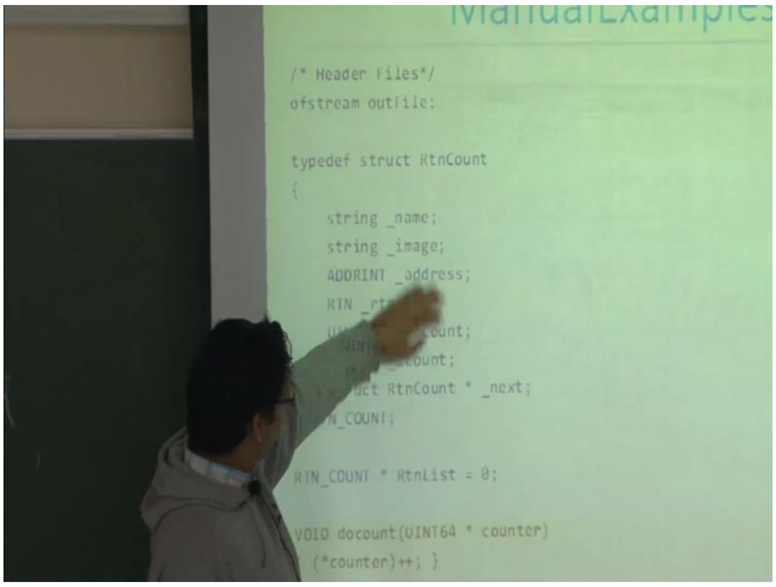
(Refer Slide Time: 33:19)



```
$ pin -t proccount.so -- /bin/grep proccount.cpp Makefile
  proccount_SOURCES = proccount.cpp

$ head [illegible]ount.out
              Procedure  Image       Address     Calls Instructions
                  _fini  libc.so.6  0x40144d00  1            21
 [illegible]ter_frame_info  libc.so.6  0x40143f60  2            70
 [illegible]ister_frame_info  libc.so.6  0x40143df0  2            62
              fde_merge  libc.so.6  0x40143870  0             8
            __init_misc  libc.so.6  0x40115824  1            85
            __getclktck  libc.so.6  0x401157f4  0             2
                 munmap  libc.so.6  0x40112ca0  1             9
                   mmap  libc.so.6  0x40112bb0  1            23
            getpagesize  libc.so.6  0x4010f934  2            26
```

So, this is a procedure instruction count this is also taken from the pin manual. So, this is the tool proc count sources. And this is the instruction that we are doing instrumenting we are doing a grip on we are trying to find this thing verifying, this is the output. So, this actually periods of file proc count and this is the output of proc count. When you seeing this thing, so it gives us the name of the function, which an prints the ((Refer Time: 33:53)) library which is providing this functions. And prints the address of the starting of the function and pin the how many call to were weight to this function and how many instructions are there in the function.

(Refer Slide Time: 34:10)



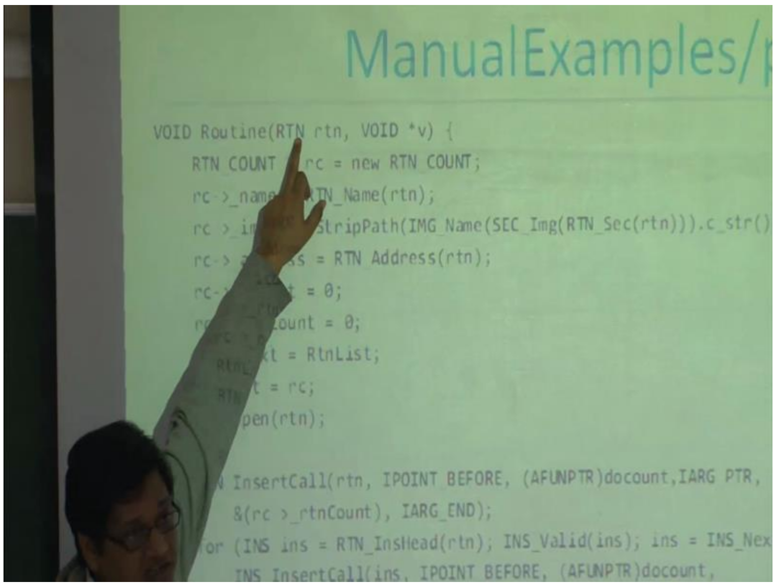
ManualExamples

```
/* Header Files*/
ofstream outFile;

typedef struct RtnCount
{
    string _name;
    string _image;
    ADDRINT _address;
    RTN _rt[illegible]
    [illegible]count;
     [illegible]count;
  [illegible]uct RtnCount * _next;
[illegible]N_COUNT;

RTN_COUNT * RtnList = 0;

VOID docount(UINT64 * counter)
  (*counter)++; }
```

So, this is the part of the pin tool, so we define some structure, which so r t n means routine which is same for the function. This structure that the name and this is the name of the image the library which is the address this is all the information that was printed over. This is the pointed to the routine this r t n, how many time it is called in the number of instructions. It is so this will we use to make a link list of the routine.

And then, we pass of pointer to it and it just increment this is our analysis routine. We pass pointer to it, to an integer and is just increments it pi 1. And this is just to get some path.

(Refer Slide Time: 34:57)



ManualExamples/

```
VOID Routine(RTN rtn, VOID *v) {
    RTN COUNT [illegible] rc = new RTN COUNT;
    rc->_name [illegible] TN_Name(rtn);
    rc->_i [illegible] StripPath(IMG Name(SEC_Img(RTN_Sec(rtn))).c_str()
    rc-> [illegible] s = RTN Address(rtn);
    rc- [illegible] t = 0;
    r [illegible] ount = 0;
    [illegible] t = RtnList;
    [illegible] t = rc;
    [illegible] pen(rtn);

    [illegible] InsertCall(rtn, IPOINT BEFORE, (AFUNPTR)docount,IARG PTR,
        &(rc->_rtnCount), IARG_END);
    or (INS ins = RTN_InsHead(rtn); INS_Valid(ins); ins = INS_Nex
        INS InsertCall(ins, IPOINT BEFORE, (AFUNPTR)docount,
```

This is the instrumentation function and we are doing r t n add instrument. So, whenever pin is about to generate code for a function, it will call this instrumentation routine. So in the instrumentation routine, we get the routine for which it is about to generate code. We create a new object of the

Student: r t n count was the structure.

So, r t n count is this structure, create a new object for this routine we initialize it is name is a pin EA functions to get the name of the routine. So, I am pin is after words this is something to get the name of the icon. This is to get the name of the library, we initialize the libraries name we get the address of the routine. We said I routine I instruction count 0, we do routine, how many time the routine is called to also 0. And we are preparing a link list of the routines, which are called. So, this is...

Student: ((Refer Time: 36:14))

R t n is the head of the list. So, we are saying routine dash to next is the head current head of the routine and we said the new head has the new a routine. So, whenever we about to instrument the routine, we should have I will talk about this operator. From it is look at this is the main instrumentation, but this was the manichery here. We are doing r t n insert call, this is similar to INS or trace insert call and we are passing routine the position I point before to the routine the analysis function is do count.

And we are passing the IARG. So, this is the input type for IARG pointer that we are trying to pass. And, we are passing a pointed to r t n count, so we adding a call to do count function, just before this routine. And we are passing the do count the r t n count, r t n count is actually trying to maintain, how many times that routine is being called. So, whenever the routine is about to called, this do count would also be called.

And it will increment this r t n count by 1. So, it will count the number of time this function is being called. So, the second thing...

Student: ((Refer Time: 37:49))

So now, we are I am trating over the routine, the instructions in the routine. So, we get the head instruction at the starting of the routine and we if the instruction is valid we keep on. So we I trating over the instruction, so a routine is just a can we thought over the link list of instruction. So, we are I trating was that link list and then we are doing instruction instrumentation.

And we are just instrumenting a call to do count before this instruction. And we are passing this I count variable of the structure as the input argument. So, whenever instruction that routine would be executed, it will counting it will increment, this i count by 1. So, this I count would actually count the total number of instructions, which are executed by the routine and all the invitations are all taken. We could have optimized this by doing similarly, breaking down it in a basic blocks and it.

So, one more thing is before doing the instrumentation we have to do this, we have to open this routine r t n open in the reports this is...

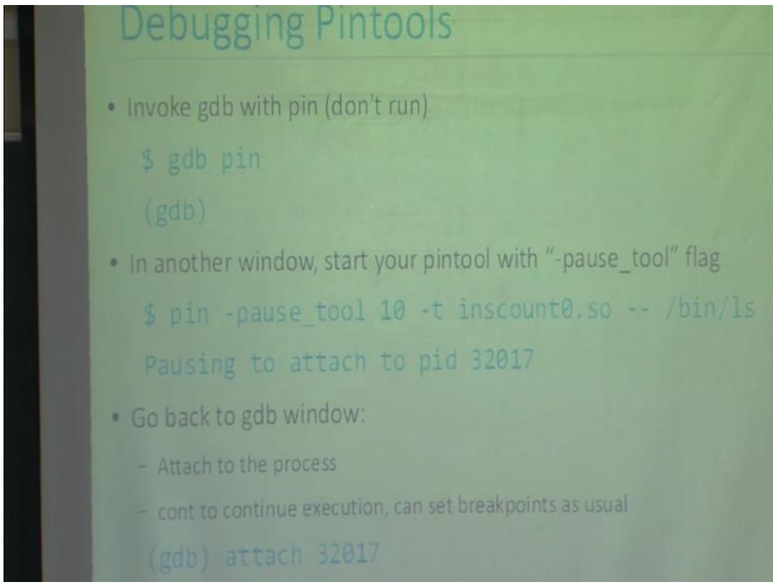
Student: ((Refer Time: 39:15) provided in the pin.

Always information is provided by pin, r t n can is there to get you can actually maintain at like this maintains the routines are structure just instructions.

Student: ((Refer Time: 39:36))

But, I did not see ((Refer Time: 39:45)) there must be for the main function in the program.

(Refer Slide Time: 39:56)



Debugging Pintools

- Invoke gdb with pin (don't run)

```
$ gdb pin
(gdb)
```

- In another window, start your pintool with "-pause_tool" flag

```
$ pin -pause_tool 10 -t inscount0.so -- /bin/ls
Pausing to attach to pid 32017
```

- Go back to gdb window:
  - Attach to the process
  - cont to continue execution, can set breakpoints as usual

```
(gdb) attach 32017
```

So, how to these the pin tools are sequence bigger than, so g d b debugging them is like a different. So, we use g d b for debugging, so where ever we yet we have to first combine them with ((Refer Time: 40:05))

Student: ((Refer Time: 40:06))

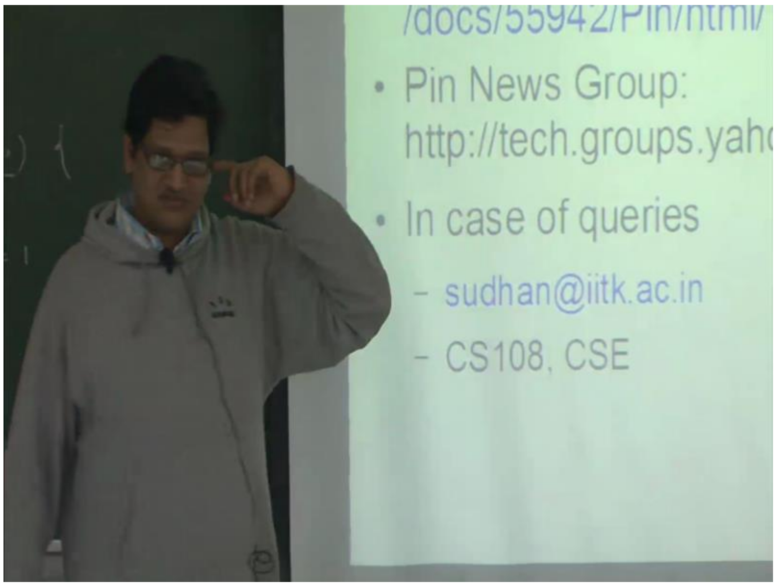
So, you first do g d b pin in one terminal and you do not run anything here. So it just you start this in and then second terminal you actually call your pin routine, but you also pause this pause tool command line argument. And, this is the number of seconds that, so what this will do is that, after starting the program it will weight for debugging seconds, after that it will start the program.

So, you can pause in it ((Refer Time: 40:57)). In this 10 seconds you have to go back this terminal. And you have to execute this. So, it will print this thing, so you have to print the p i d of this process. In this terminal you have to do attach to this p i d. So what it will do it will attach this to this terminal, and then you can do all break can insert.

Student: ((Refer Time: 41:126))

So here, after you do attach you can a create break point a continue. And when you do this it will also print a line which says which the common and which is to, so to add the simple information of the pin tool in with g d b. So, after doing attach you should execute that common I am not showing that here. And it will attach the simple information was and then you can insert as usual break points. So, this is the last one.

(Refer Slide Time: 42:13)



/docs/55942/Pin/html/

- Pin News Group: http://tech.groups.yah
- In case of queries
  - sudhan@iitk.ac.in
  - CS108, CSE

And, this is the link to the pin manual you can also find it on the home page of pin. You should go through it. There are not more examples in it, you should detail. Then, there is a pin news group, if you have any questions you can ask them, you can post on this. And you can win if the question you can ask me this is my mail id and this is my ((Refer Time: 42:35)).

Student: Thanks, enough.