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

Lecture - 8
Binary Instrumentation for Architectural Studies: PIN

Previous tutorial, we were seen what ((Refer Time: 00:16)) Instrumentations and what dynamic binary instrumentation means? Then, we would basically see how pin actually does dynamic binary instrumentation. And how to use pin for architectural study? So, you guys would be using pin for your homework assignments. So, please pay attention. And if you have any question, please feel free to interrupt me.

So, let us start with instrumentation. So, instrumentation is the technique for inserting extra code into an application to observes study and even change it is behavior. Lot of program analysis, screws and instrumentation for performer profiling error detection and capture and replay ((Refer Time: 01:09)) user instrumentation for memory error analysis. It can detect misuse of mellocks and free. So, by misuse of mellock, I mean that you have allocated some memory segment using mellock.

And you are not freeing when it loose ((Refer Time: 01:27)). So, it leads to memory leads. So, other can identify those. By misuse of free, I mean that you are freeing some unallocated memory segment. So, it will lead to segmentation font. It can even detect uninitialized variables and dangling pointers. You guys know, what dangling pointers are?

Student: Pointers are not initialized ((Refer Time: 01:55))

Dangling pointers are, actually point to a memory region which has not been allocated by the application. So, dereferencing can be lead to segmentation font. Intel specter is also uses instrumentation for thread error analysis. It can detect data registers in your multi thread application and even in deadlocks and multi threaded applications. Intel's v tune amplifier also uses as an instrumentation for performance analysis.

And it can locate hotspots in your application. And it can even do a lock weight analysis. So, it can give you the lock contents in time. We would be mainly using instrumentation for architectural study. We would be doing processor, branch predictor and cash

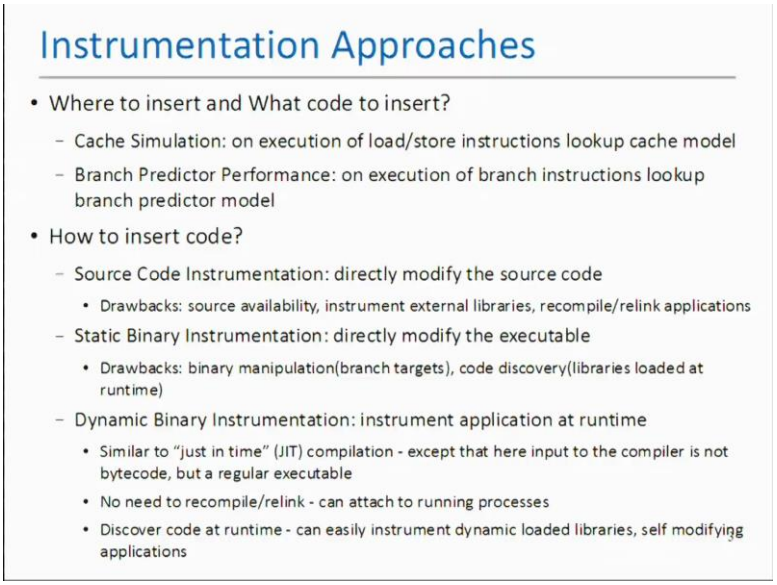
stimulation sending instrumentation through. And you are using instrumentation to collect instruction crisis. And these instruction crises would then be failed to some other trace base stimulator, detail stimulator.

So, Intel software development emulator also used as a instrumentation to emulate new ((Refer Time: 03:09)) instruction. Suppose that, you want to implement some new instruction and study what benefit you would get, because of it. So, what you do is that? You register with software development emulator, function which implements the functional behavior of the application.

And when software development emulator encounters this new instruction, instead of trying to execute it natively, it will actually call the function that you have register to emulate it is behavior. So, now moving on to instrumentation approaches. So, then we are doing instrumentation, the two main questions that arise are, where do we want to insert code? And what code do we want to insert? So, these two questions are mostly related to the problem, that we are trying to solve.

So, if you are doing cash stimulation, then one execution of each load store instruction, we would want to look up a cash model that you have developed. So, in cash stimulation, so we are inserting code at execution of a load store instruction. And the code that we are inserting is to look up a cash model.

(Refer Slide Time: 04:22)



Similarly, in a branch predictor performance study, we would want to insert code. Just before the execution of a branch instruction. And it would look up the branch predictor that we have preferred and developed, whether there is a branch or ahead? And the next question that arises, how do we actually grow about in inserting the code? There are various approaches to it. We can directly modify the application source code.

And we could directly write code, add code in it. This is called source code instrumentation. It is not, I mean you must have used it for measuring some time taken by particular chunk of application. You add code before that chunk to measure the starting time. And then at the end of it, to measure the end time to get the time, that is going to be a particular chunk. It is not deadly used. We would suffer from major drawbacks you.

Because, you are doing source code instrumentation, you should have the source code with you. Just modifying, just instrumenting the source code of the application is not sufficient. You would even want to modify the source code of the third party libraries at this ((Refer Time: 05:32)). So, why? Can anybody tell me, why modify the third party libraries at the application that is using? Think about cash simulation.

So, you are trying to add, insert code. Just before it, each load store instruction, you look up the cash model. So, your library function that you execute maybe doing some load store, maybe executing some load store instruction. So, you would want to insert code to look up the cash model before them also. So, just instrumenting the application source code is not sufficient. You would have to instrument all the library that it is using.

And this, finally it requires recompilation in relinking the application. The second technique is to do static binary.

Student: Just one minute. So, can you think of like one. So, yes he has talked about the drawback, source and everything in it. So, why is it a problem? Does it awkward to you, that is a problem. Source availability and it is so why? Somebody?

From most of the commercial application, we do not have the source code.

Student: So, do we have the source code or Microsoft word? How many has that, anybody? Instrument, Microsoft word like that. This should be integer.

I mean most commonly used applications are some of these common windows, commercial applications. We do not have the source code for them. So, it is difficult to instrument them through source code instrumentation. The second technique is static binary instrumentation. Here instead, it is similar to source code instrumentation. But, instead of modifying the source code, here we directly modify the executable binary of the application.

It has a couple of similar drawbacks. You need to know, what is the binary format? Because, you have to modify it. So, we are trying to insert some extra code in the binary. So, what would happen is that? The instruction addresses may change, because of this extra code that we added. So, now this is a branch instruction. The branch target we would even have to change the branch target.

So, the starting extra code is not there. In the original, you have also needs to be modified. Then, both discoveries another problem which static binary instrumentation suffers from the code discovery I mean is that, most of the applications at we use, this use the dynamic libraries. So, the third party libraries that we used. The code for the third party libraries are not in the application banded. It is loaded in the application time. It is loaded at run time.

So, when you are doing static binary instrumentation, we do not have the code for the third party libraries in the key word. So, here also you would have to actually look for the binaries of these libraries and modify them also. So, here also just modifying the application binary is not sufficient. The third approach which is most commonly used instrumentation approach is dynamic binary instrumentation.

And dynamic binary instrumentation, we instrument the application at run time. As the application is executing, you insert some extra code in it. And you execute it along with the application. It is similar to just in time compilation. So, are you guys familiar with just in time compilation? Do you know any compiler with

Student: Java.

How it works?

Student: ((Refer Time: 09:31))

They changes, what?

Student: Binary code change. It changes the binary code. And then, that binary code is compiled at that wherever you getting.

No, so that is not just in time. That is the java compilation. So, when you compile a java program, it generates byte code and gets. So, the class files contain byte code. Now, when you run this java program, with the java works like. So, you write java and the class name. So, when you run it and so java is the java virtual. So, when it starts executing the program, it reads the byte code from the class files.

And it generates machine code for it, corresponding to it. And then, the generated machine code is actually what is executed. So, it keeps on reading machine byte code and generates machine code. And then, execute it because process keeps until the application ends. So, dynamic binary instrumentation is similar to just in time compilation. Except here, instead of translating from byte code to machine code, you are translating from machine code to machine code.

And while it is doing this translation from machine code to machine code, you have an opportunity to insert some extra code in the application. So, while this translation is going on, it acts extra code. And then, the generated code actually has the original code and it has the extra code that you have avoided. And then, it executes it.

Student: Do you have the machine code into ((Refer Time: 11:03))?

I mean, finally machine code would not run. So, you would see how it works? So, the benefit of dynamic binary instrumentation is that, you do not need to recompile and rename the application. You can, because everything is happening at run time. You can directly attach your instrumentation to use to a running process. So, most of the tool that, instruments are like web servers and database server, which are demon processor.

So, you can directly attach to them.

Student: ((Refer Time: 11:38))

So, here the code discovery also becomes much easier. Because, if your application is using some dynamic loaded libraries. Then, the library code would also be loaded first in

the address space. And when it is about to be executed, at that point the instrumentation engine can insert code in it also. So, just like a static binary, we have to explicitly go and look for third party libraries and add code.

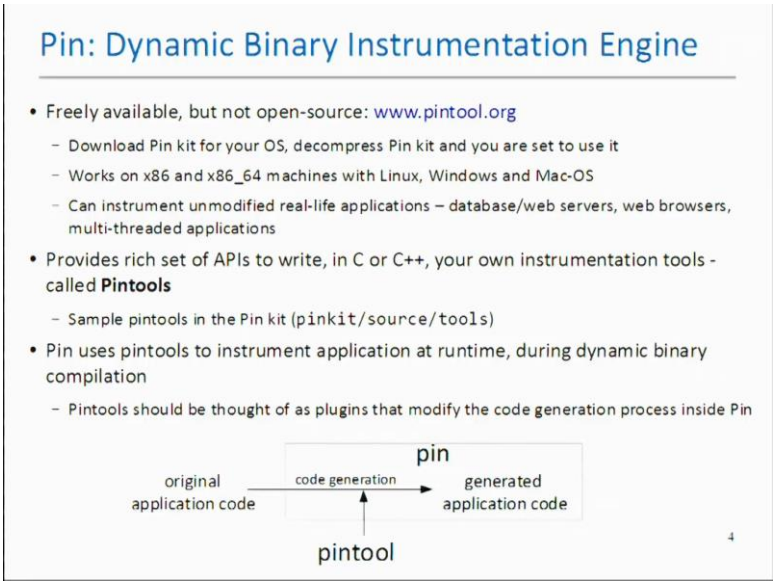
It is not necessary here. And this event takes care about of self modifying applications. So, self modifying applications modify their own source code after their executing. So, that would also be taken here by this. We would see how this works after sometimes? Let us move to pin. So, pin is a dynamic binary instrumentation image. Engine, it does dynamic binary instrumentation. It is not free. It is free available. But, it is not open source.

You can go to this web site. Pin tool, that all you can download the pin kit for your os. You will get a compressed archive, decompressed. There is nothing to install. As you decompress, you are ready to music. It works on x 86 32 bit and 64 bit platforms and on the major operating system in a Linux, window, macro. The macro stores support is relatively new. I do not know, how it will works?

It can instrument, unmodified real life applications like database servers, web servers, web browser and multi threaded. We can run an application on any of these fitting systems, then you can instrument it, that is it. So, in pin kit you actually get a binary pin pin, a binary of the pin instrumentation in engine. So, pin instrumentation engine exposes a set of API. Using this API, you can write in C or C plus, plus, your own instrumentation tool.

And these instrumentation tools are called pin codes. There are a lot of sampled pin tools provided in with the pin kit. This is the path name of the, this is the directory in which the sample pin tools are. We should go into them now.

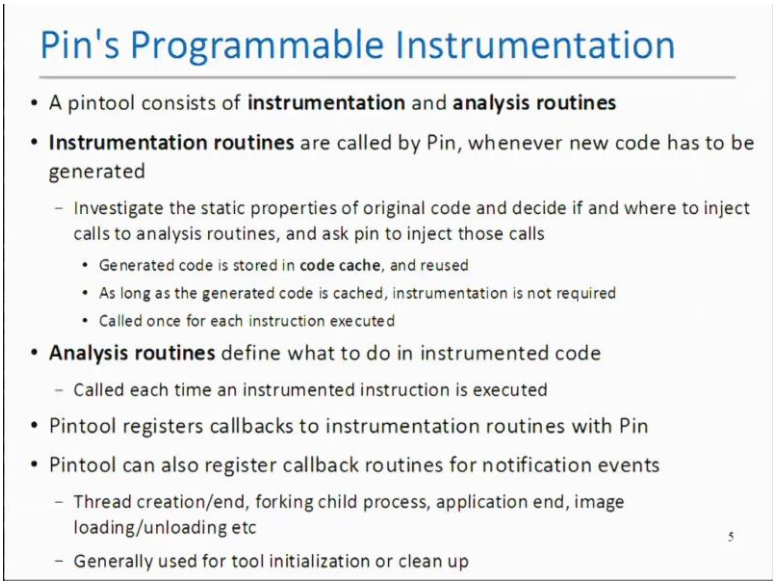
(Refer Slide Time: 14:04)



So, as I told previously that pin is a dynamic binary instrumentation engine. So, what it is doing is that, it reads the original application code. And it is generating some code for this corresponding to this application code. While it is doing this, it uses the information from the pin tool to modify this generation process. And what it will do? It will insert some extra code, which pin tool will tell. So, pin tool will tell where it wants to insert extra code?

And what code it has to insert? And pin could actually do that for the pin. So, pin uses pin tools to instrument applications at run time, during the dynamic binary compilation and pin tools can be thought of as a plugins, which modify the code generation process of pin.

(Refer Slide Time: 14:52)



So, are you clear about the…

Student: Just in time compiler, in this pin?

Yes, it is similar to a just in time compiler. Just in time compiler, you are generating machine code corresponding to some byte code. Here, you are generating machine code corresponding to…

Student: So, this is also generated by the time, the process is executing.

Yes, it will do all these thing at run time. When, the application is executing at that point, it could be executing the application and at the same point, it would be adding some extra code in the application, and that executing the extra code also. So, pin tool consists of instrumentation and analysis routines. These are simple functions. And instrumentation routines are actually, instrumentation routines are called by pin, whenever new code has to be generated.

Whenever pin is about to generate some new code, it will call these instrumentation routines. So, when an instrumentation and instrumentation routine actually investigates the static properties of the original code and the sites is somewhere to inject calls to analysis routine. So, what it is doing? It looks through the code and it decides, whether I want to inject some code over here. And it will inject calls to analysis routines.

Both of these are contained in the pin code. And so once it has decided at, it has to insert some call to analysis. It will ask pin to insert that call. So, when the code is generated, it is stored in a cash and it would be in use. So, if you have for loop, which has some number of iterations. So, the first time when the first iteration is being executed, at that point pin will generate the code for this, for the first iteration.

And while it is generating code for the first iteration, it will instrument it also. So, when this first iteration has been executed, for the second iteration it would not again call the instrumentation routine to generate code, because it has already cashed that code in the code cash. So, this using cash to stole the generated code is an optimization, which is just in time compilers view, to reduce the cost of code generation. Actually, this speeds up the process.

Student: So, just to clarify, this cash is not the hardware structure.

Yes, it is software cash.

Student: It is a border memory is awkward is storing these.

So, as long as the generated code is there in the code cash, instrumentation is not required. It can directly execute the code. So, for some application code which has already been instrumented, the generated code is present in the code cash. So, if that code throughout its life time remains in the code cash, the instrumentation routine would not be called again for it. So, most of the time, this code the size of code cash is sufficient to cash the whole application.

So, instrumentation function would only be called for the original code once. So, here called at most, once for an application.

Student: Can you configure the size of the ((Refer Time: 18:22))

So, these are the instrumentations. And so we were injecting calls to analysis routines. So, analysis routine actually defined, what the instrumentation code would do?

(Refer Slide Time: 18:40)

So, this is the original application code. For this, the generated because we are translating from machine code to machine code, the generated code would also be almost similar to it. So, instruction would remain same. The register allocation might differ. But, I have behalf inserted a call to analysis one. So, this is the original code. We have, the pin has generated this code for it. And the pin tool has instrumented this code.

And injected are calls to analysis one function. So, each time this compare instruction is about to be executed. Just before this compare instruction, this analysis function would also be executed. So, each instruction which had been instrumented, the analysis functions will be executed each time. That instruction is also executed. Do you get this?

Student: So, instrumentation routines are caring you, what instruct can be an instrument?

So, you can where to instrument?

Student: And you can put it after instruction, before instruction.

Yes, we will talk about that.

Student: You talk about that. Analysis routine is telling you, what to do? So, the where and what here is injecting call to analysis routines. And the where is decided by the instrumentation routine.

Student: Instrumentation means only inserting calls.

Yes, and these are these instrumentation function would be called, at most of the time buns for these instruction. And analysis functions, we called each time that instruction is executed.

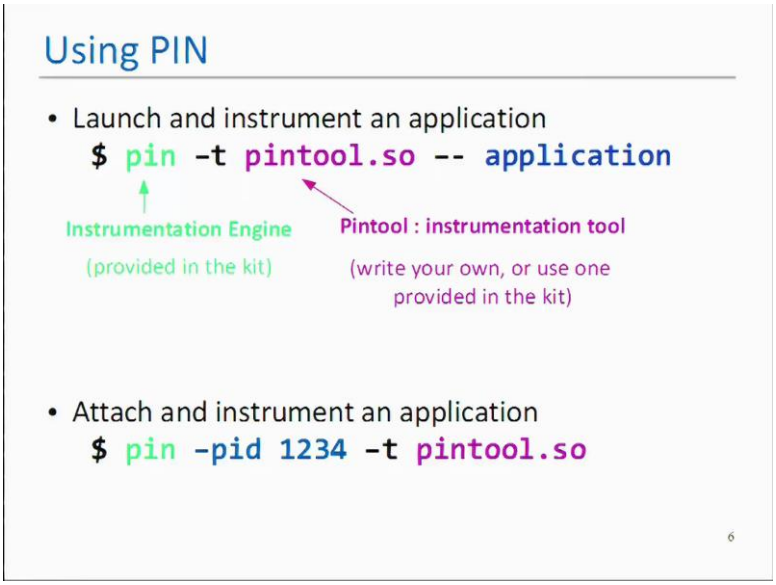
Student: So, just one more comment. So, you can see that, the instrumentation in the best possible case should be evoked, exactly ones for each instruction. Actually, that you instruct for every instruction. So, provide of course your spacing code cashier all these things. But, this analysis routine for a particular instruction will get invoked every time the instruction executes.

Because, the code has been some generated code contains a call to this analysis routine. So, each time it traverses this code path. This analysis function would be called. So, analysis routine could be called each time, it traverses this code. So, pin tool consists of instrumentation analysis routine. But, so pin tool also registers with instrumentation routine, pin has to call. So, it registers a call back to the instrumentation routine, which same you should use.

And pin tool, so just like it was registering call backs to instrumentation routine. Pin tool can also register call back call notification event. Call backs to routines for notification events. These notification events can be thread start, thread end, fork of a child process, forking a child process. Or this can be the application end or image loading and unloading. Here, image refers to a library which has been binary file, which is been loaded.

So, whenever a binary file for a dynamic library is loaded, we can call a function. We can register a call to a function. So, these are actually, this event notifications are generally used for initializing some part of the pin tool and ((Refer Time: 22:29)).

(Refer Slide Time: 22:30)



So, your pin tool is a C, C plus, plus program, you write it you compile it you create a dot, dot so file, for a library. Is this visible? So, you compile it. You get a pin tool dot so file and you instrumentation engine is provided with a pin kit. So, you supply the pin tool to the instrumentation engine as a command line argument. And after this double slat, you supply the application that you want to instrument.

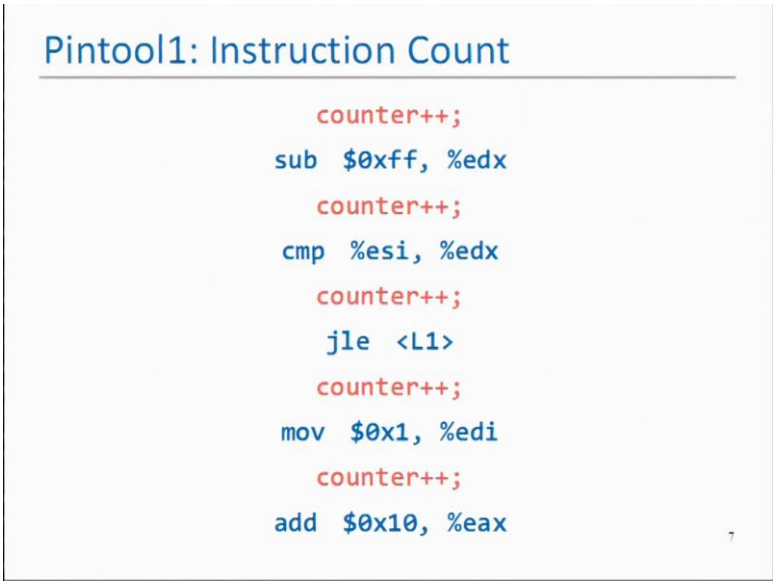
Student: That how can you find?

The application binary. So, this pin tool either you can use the pin kit sample tools or you can write your own. And you can attach pin to already running process. For that, you have to supply the pid of the process, with the dash pid command line.

Student: So, you are going to be with some examples of pid.

Yes, after this we have many examples. So, you get this how to…

(Refer Slide Time: 23:31)



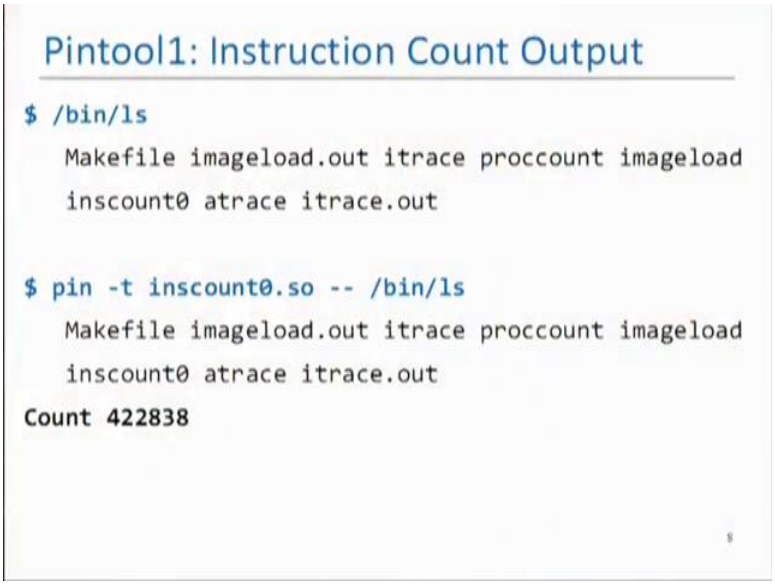
So, let us look at. You are trying to write a pin tool, which actually count the number of instructions that have been executed. So, the code the assembly code here is actually the applications code. And what we want to do is that, we want to insert these counter plus, plus. We want to increment the counter, when insert code to increment this counter. Just before each instruction. So, whenever this instruction is about to be executed, just before it the counter would be incremented.

So, in this way you can maintain the number of instructions that are executed.

Student: So, just one question. So, in that example in most cases you will try to implement the analysis function.

So, it was inserting calls to analysis routine. So, here number of analysis routine would consist of incrementing this counter. But, pin can. So, compile this pin tool. So, it also does compiler optimization. So, at there it will see that, this is a simple function. So, most of it will inline these functions. So, this is what we want to do.

(Refer Slide Time: 24:48)



Suppose, that we have written this pin tool. And so we want to instrument l s. So, this is what l s normally prints. When we run it with pin like this, this is our tool inscount zero dot so.

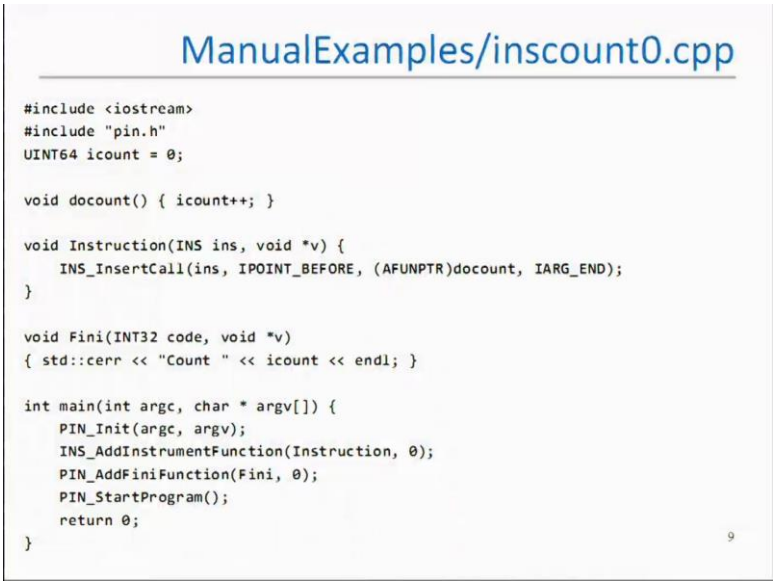
Student: No, wait I see. So, you are running l s directory, which has these.

Yes, which has these files and when we run it with pin, it gives the, this is the output from l s. And this is what, pin tool with output. So, it will count… It will give us the number of instructions that the l s command is executed. Does everybody follows, this one? So, this is the pin tool and by after doubles that, this is the binary that we want to instrument.

Student: So, you are going to show the pin tool.

Yes, of this…

(Refer Slide Time: 25:33)



So, let see. So, this is the code for the inscount pin tool. It is a simple C plus, plus application. This is the file in a pin kit in the source tool folder. You can take from this file. So, let us see the source code. It has some header files. We include the pin dot h header file, because we are going to use the pin API functions. We define this 64 bit integer value. This is the counter which is going to keep the count for number of instruction, which it executes.

We initialize it to zero. Then, we have this do count function, which increment this counter by 1, whenever it is called. Then, we have this function instruction. It takes some arguments and this calls this instruction INS insert call function. So, this is the pin API function. We would talk about this input argument. But, let us see the third argument. Third argument is a pointer to the do count function. So, the third argument is the pointer to the do count function.

Now, the other function is the Fini function. We will talk about, what are the input arguments are. And it just prints on the standard error std::cerr, the i count value. In the main function, so this is your pin tool code. In a main function, we have code pin API calls. The first one is PIN error. Then, is INS_Add instrument function, this takes a pointer to the instruction function. So, this instruction function is defined over here.

And then, there is PIN_Add Fini function. This takes a pointer to the Fini function, which is defined in the pin tool. So, we are using function pointers quite a lot. And then,

there is PIN_StartProgram. Now, so PIN_Init actually takes the command line argument. You can pass some command line arguments also to the pin tool. So, here we are passing the arguments to the PIN to initialize the command line arguments and run time system.

Student: But, can you overwrite the Init function?

No, you cannot. You have to only, so your pin tool would be… This would be the first function that your pin tool will execute. This has to be always present.

Student: So, its internal cannot expose. You cannot do with that, fine.

You cannot modify any of these functions. Now, actually we cannot here.

Student: Actually is to get a function pointer. There you do not have any freed out all.

So, yes. So, INS_Add instrument function is… So, it takes argument as the. So, this is our, so through INS_Add instrument function, the pin tool is registering with pin. What is the instrumentation routine to call, when it is about to generate code? So, this says that, whenever PIN is about to generate code for an instruction, it should first before generating the code, it should call the instruction function.

So, this is the instruction function is the instrumentation routine. It would be called each time PIN is about to generate code. So, let see what we are doing in the instrumentation function? The instrumentation function takes as the import or reference to the instruction for which PIN is about to generate code. And the second argument, here is the argument that is passed here. So, this zero ends pass in this.

Student: No, we want to mention that INS ((Refer Time: 29:21)). You are looking about this.

So, this is registering whenever PIN is about to generate, code for an instruction. It should call this instruction function before that. And the second argument here is, this 0. So, we not use this much. So, most of the time, it would be 0. Now, let see would it, what we are doing inside the instrumentation function? Here, we are calling this in INS_InsertCall. Here we are, so we want to insert call to this. We want to increment the counter at each before each instruction.

So, we are saying that before this instruction, we want to we are asking PIN to insert a call to this docount function, before it generates, before the code for this instruction.

Student: As the second order

This is the, coming to that

Student: You just mention before that. That is why I am saying.

So, we would talk about this IPOINT_BEFORE, leave it for now. And this IARG_END, we will talk about this. So, INS_InsertCall is asking PIN to inject a call to do count function, before it generates code for this ins function.

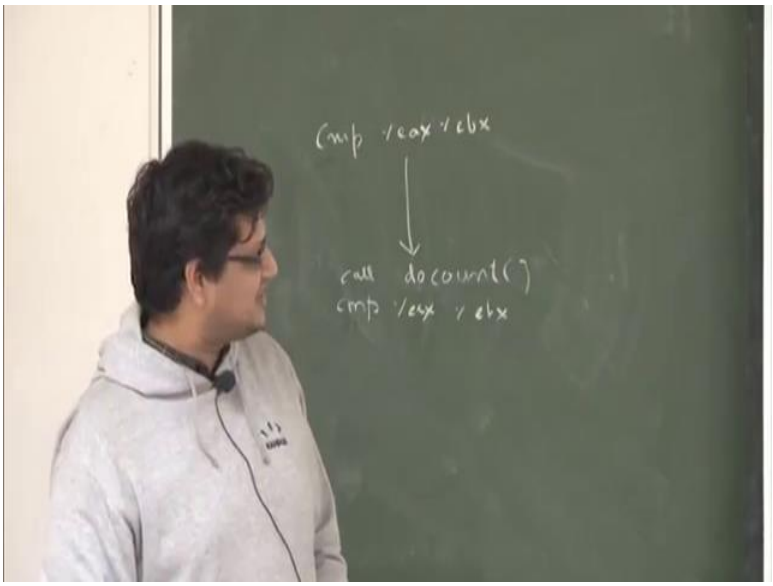
Student: This instruction computed can be called anytime.

No, this instruction function is the instrumentation function. It would be called by PIN, whenever PIN is about to generate code for an instruction. So, ((Refer Time: 30:56)) suppose that PIN is about to generate code for this compare instruction, so before generating code for it, PIN will call the instruction function. The instruction function says that, insert a call before you generate code for this function to insert a call to this do count function.

So, call do count and then PIN will generate code for this instruction. So, this here…

Student: Are does it no going to call this function?

(Refer Slide Time: 31:32)



Using before doing instrumentation, otherwise no instrumentation that is I cannot…

Student: That INS prefix is not it telling it, that you should do this for every instruction.

I would come to that, I am saying that INS instrument add function is saying that, whenever it is about to generate code for a instruction. It should call this instruction function first. And this instruction function is the instrumentation function. It is asking PIN to insert a call, inject a call to do count function, just before it generates code for this particular instruction. Do you have any doubts here?

Student: Can I say this function after an instruction instrument, this code and another function after this instruction ((Refer Time: 32:28))?

I did not get you.

Student: He wants to add two instrumentation functions.

You can have as many instrumentation functions. You can have another instrumentation function, which maybe inserting call to some other analysis routine. Or you can insert call to the second analysis routine here only. You can do another INS_InsertCall some two count to.

Student: But, I want to add this call to specific function. Is it possible?

Add this call to its specific function. So, you are saying that, suppose that whenever print f is being called.

Student: You can.

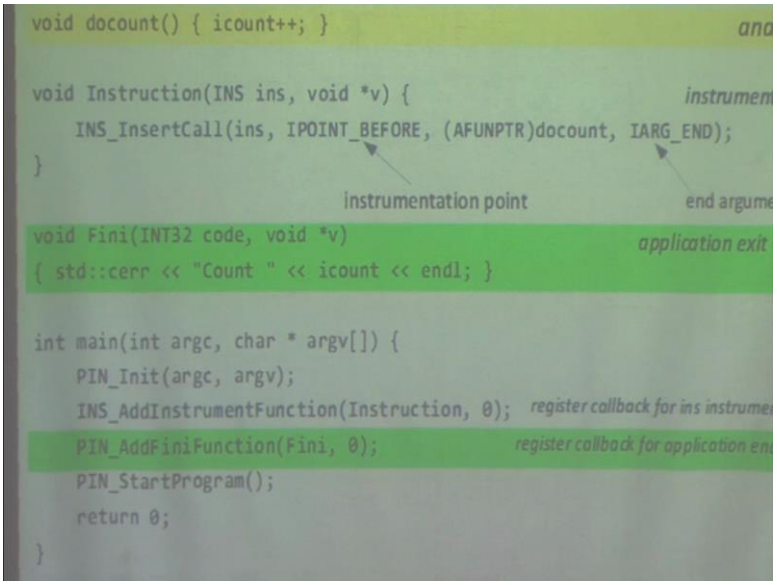
So, till now we are not analyzing, what this instruction is. This instruction maybe, calls to printer. We can even analyze that, I will come to that. So, this is just introducing what instrumentation analysis routines are, how? Do you understand this?

Student: So, the just one comment here is that, what we are going to mention to that ((Refer Time: 33:25)). The purpose of Fini is to…

I am coming to that.

Student: You come. You come to that.

(Refer Slide Time: 33:37)



So, the do count is the analysis routine. How it is focus on this PIN_AddFiniFunction. Before that, so this is your instrumentation point. In the next slide, I will talk about what is the instrumentation point? And this is your end argument to the insert call. We will come to that also.

Student: That is some marker, is not it?

Yes, some marker now. So, I told you previously that PIN can also register call backs to event for events, call back to routine for events. So, PIN_AddFini is registering a call back for Fini function, whenever the application is about to exit. So, it registers a call back for application. Whenever the application is about to exit, it will call this Fini function. So, that is what we want to do, whenever the application is upon the print, the number of instruction that have executed.

So, this function just prints count and the instructions that are executed.

Student: Is it a normal entry, z minus 1?

Yes, at that point also this would be common.

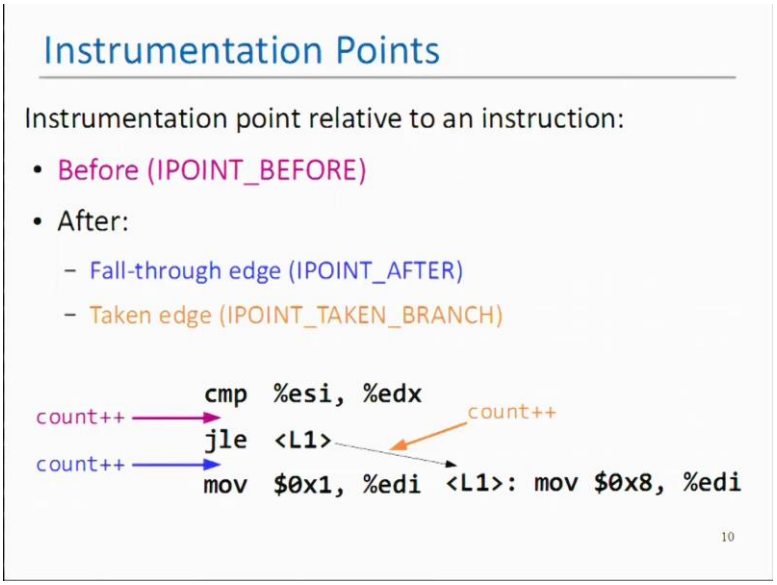
Student: So, can we use this function for any other purpose, whatever we want to do at a termination essentially.

So, registering a count ((Refer Time: 34:48)). Here, we are counting number of instructions. So, we just print the count. I mean, we have allocated some exactly memory for your pin tool. You may free it at application end. These event functions are generally for initializing and clean up at the end.

Student: Actually, here I could use C out also or is a problem?

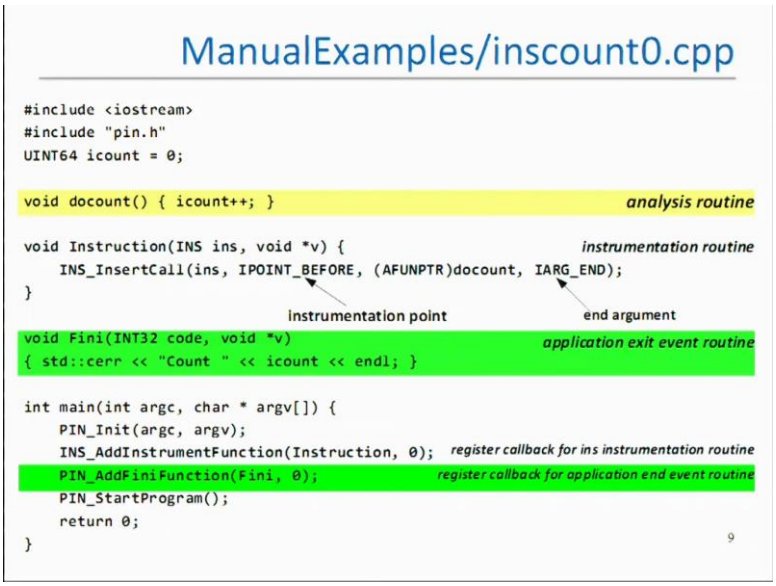
You can use C out also. Not included in instrumentation.

(Refer Slide Time: 35:21)



Student: At a last call, essentially gets start about that. Forgot that?

(Refer Slide Time: 35:25)



So, when this function… So, here we were initializing the run time system. Here, we just registered with PIN to call this instruction function, when it is about to do instrumentation. And in this PIN_AddFiniFunction, we are registering a call to the Fini function, when it is about to end. So, pin this point, the application is not started neither as the instrumentation. And when we do pin start program, so this function never returns actually. So, this return zero is just for compiler.

Student: So, it is a long jump, is not it?

So, this return is just to make the compiler happy. This never returns.

Student: So, that one is a long jump.

Yes and it basically, when this function execute it starts the application and starts the instrumentation process. So, these two functions would always be there in your pin tool. And you would add instrumentation, routines and notification modules. Now, let coming to the instrumentation point. So, if you remember that, here the second argument was the instrumentation point. So, this Insert Call was taking the instruction reference as one of the first input.

Then, an instrumentation point and the third argument was the analysis routine. So, the instruction point actually, it refers to a position relative to the instruction. So, if we look at this j l e instruction ((Refer Time: 37:00)). So, that three positions, so it says. So, whenever we are inserting call to analysis routine. So, we can say whether we want to insert call to analysis routine before this function, before this instruction or after this instruction.

So, before this instruction, so we give we had in the last example, we had given IPOINT_BEFORE. So, it will insert the call to do count function over here, corresponding to this j l e instruction. After at their two positions, because this is a branch instruction the fall through h. The fall through h follows is the next instruction in the source. And the target is this edge is called the taken edge.

So, for the fall through edge, we could use IPOINT_AFTER. And for the taken edge, we can use IPOINT_TAKEN_BRANCH. So, if here we would have given IPOINT_AFTER, then what would have happen is that, we can go to this. If we have given IPOINT_AFTER, ((Refer Time: 38:15)) then this counter plus, plus would have come here. It would have been inserted here. ((Refer Time: 38:21)) So, this is clear to everybody?

So, this is a problem only with branches, thus we see here. But, after branch there to be two possibilities, depending on which way the branch goes. So, accordingly so both of these may not be even defined for some instructions. So, if this is unconditional jump. So, there is no fall through edge after edge. So, because it will always jump, so IPOINT fall through edge. IPOINT_AFTER is not valid. In this, this is a conditional branch.

Unconditional branch only the taken edge is valid. So, if this is not a compared, if it is not a branch instruction, like this is compare instruction. There is no taken edge here. There is only fall through edge here. So, these two may not be defined for each instruction. But, IPOINT_BEFORE is always defined for an instruction.

Student: Most of this in the previous example, instead of IPOINT_BEFORE I specified IPOINT_AFTER. Will, so that may. So, it will compile correctly.

Yes, it will compile correctly.

Student: But, so it will not be able to count the taken branches.

That would, so you would have to actually somehow modify your instrumentation course.

Student: So, what if I specify it both?

You would have to specify both to add.

Student: So, in that case will they be double counting from non branch instruction?

No. wait, because the other part does not exit so.

Student: You understand what sir was saying.

Suppose that, in this example here we do not want to insert call at before the instruction. We want to insert it after the instruction. So, if we give only IPOINT_AFTER, so what will happen is that, for conditional branches I mean this counter plus, plus would never be executed. If the branch is taken for conditional branches, if the branch is taken then the counter plus, plus which is just after, it would not be executed.

So, we would insert the counter plus, plus on both these paths, IPOINT_AFTER and the taken branch.

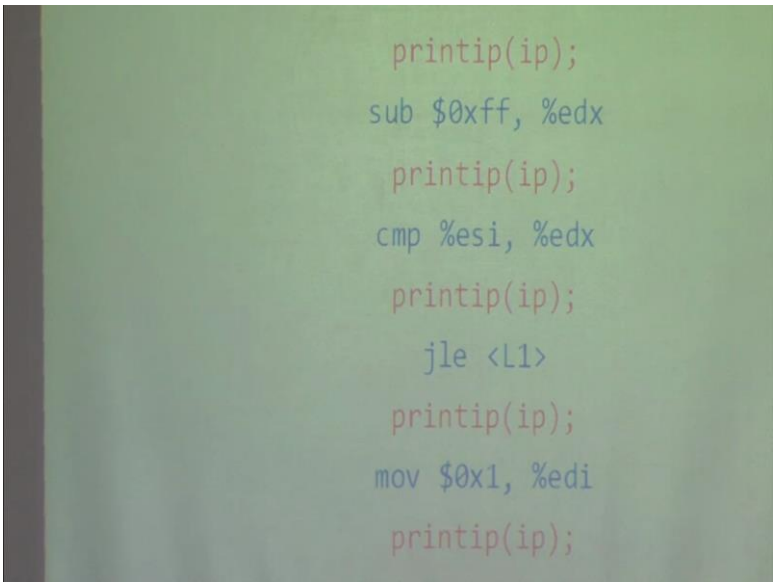
Student: So, maybe I clarify one thing. We talked about this branch devious last class 10. It does not have anything of. So, here there is no devious. Just everybody clear?

So, most of the time we would be using IPOINT_BEFORE ((Refer Time: 40:50)), but some for branch predictor study, we may have to use these, all use these.

Student: So, for example, if you want to know just given a branch, how many times you get a fall through? How many times you get a taken path? There these two will be very important. Otherwise you can of course use that.

No, we can even do that with IPOINT_BEFORE. So, IPOINT I would. So, now let us look another example. So, this here we want to print the instruction trace. By instruction trace, I mean the instruction address for the instruction which are executed.

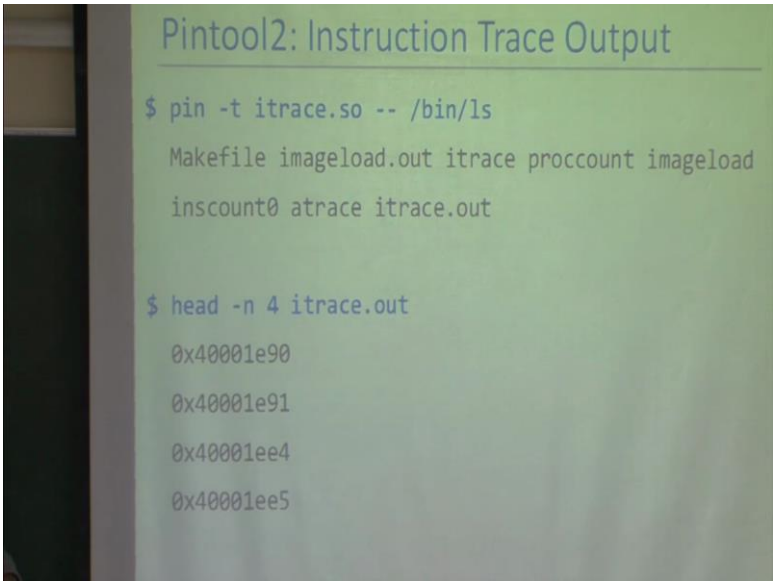
(Refer Slide Time: 41:36)



So, we would insert call to this printip function and we would supply input argument as ip. So, here we need to pass an input argument to the analysis. In the previous case, we were not passing any input argument.

Student: So, in the just one clarification, just about acronyms. So, in Indian world ip refers to Instruction pointer same as program counter. So, it is just printing the instruction pointer of every instruction, this particular pointer. So, this tool also works in the similar fashion.

(Refer Slide Time: 42:10)



This is how we will use it itrace dot so. You will write a program itrace dot cpt for it. You will compile it to get itrace dot so. And it will, it is just printing the… It is not outputting anything on the standard output or error terminal. Basically, rights in this itrace dot out file, this pin tool. So, when you run it, it will create this itrace dot out file. And this is, I am showing some part of the itrace dot out. So, it will print the instruction addresses in.

Student: So, in the first four instructions.

First four instructions.

Student: Well, first four instruction addresses. ip is internal to pin. pin defined printip.

Print ip is the instruction pointer, which pin will provide it. It will manage, it will compute ip. It will instruct to the pointer some out. And printip is the function, we would analysis function that you will supply.

Student: So, here just one minute. Go back to the trace. So, the first instruction is 1 byte. Is not it, 90 91.

First instruction is 1 byte.

Student: Second one is three large

Second one is 3 byte.

Student: Second one is pretty large, 91e4.

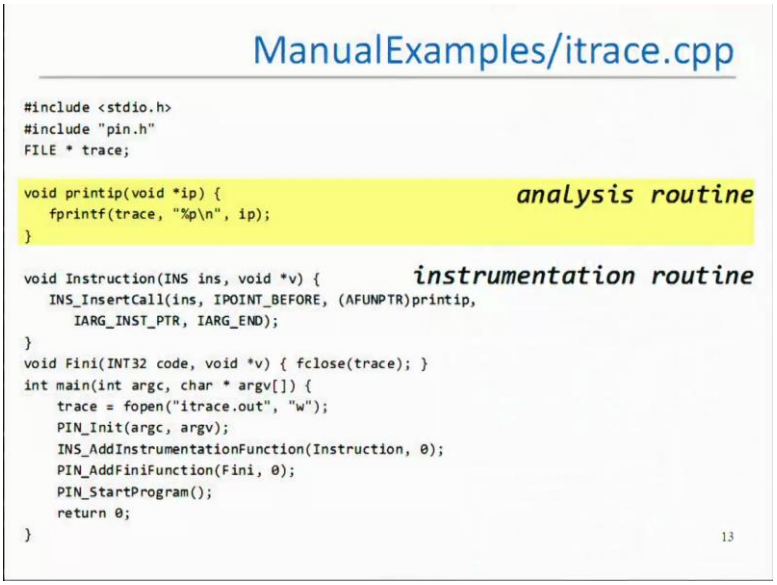
This maybe a jump.

Student: This could be a jump ((Refer Time: 43:30))

So, x86 instructions are not of the same size. They are variable size instructions.

Student: It is hard to say whether it is a jump or the other.

(Refer Slide Time: 43:46)



So, this is the code for itrace. This is also taken from the sample tools. So, here the analysis routine is printip. So, it is taking an input argument ip. And it is just printing this ip. Here this code is almost same except, that we have creating a file itrace dot i. When the program starts and in the Fini, we are just closing this file. And the instrumentation registration is almost same. The Fini is almost same. So, look at the instrumentation function, this has changed.

So, here we have some more arguments to the passing some other arguments to this INS_InsertCall function. We are passing IARG instruction pointer. So, this is these are… So, what would happen, would I here we are saying that. So, we asking PIN to insert a inject a call to printip function. And we are asking it to pass IARG instruction pointer as a input to the printip function. So, it will IARG instruction pointer is defined by the PIN API.

It actually gives you the instruction for the current instruction, whatever it is instrumenting.

Student: Global environment IARG, can I tell you it is a, it has to be right, what is it?

Yes

Student: Main, can I access it. IARG instruction pointer in main, no we cannot

So, this is kind of a I would say hash define.

Student: ((Refer Time: 45:45))

So, thinking if just gets, this is the short form I mean I would. So, basically PIN provides a API function, ins address to that function which we provide the instruction reference as a input. It will give you the address.

Student: It is a macro

It is a macro, IARG instruction pointers are macro. And it is just saying that, inject a call to printip function and give this instruction pointer as a input to it.

Student: So, this analysis function can have any number of arguments?

Yes, this analysis. So, analysis function can have any number of arguments. And we can. So, in the INS_InsertCall is specify what all the arguments we want to pass. The first three arguments to these always, most of the times are since. The first one is the instruction. The second one is the instrumentation point. The third one is the analysis function that we want to inject. And after the analysis function, we can specify any number of arguments.

We want to pass to the printip analysis function. And IARG_END, actually the macro which specifies the end of the arguments to INS_insertCall. So, whatever is between printip and IARG_END would be passed as input to printip function.

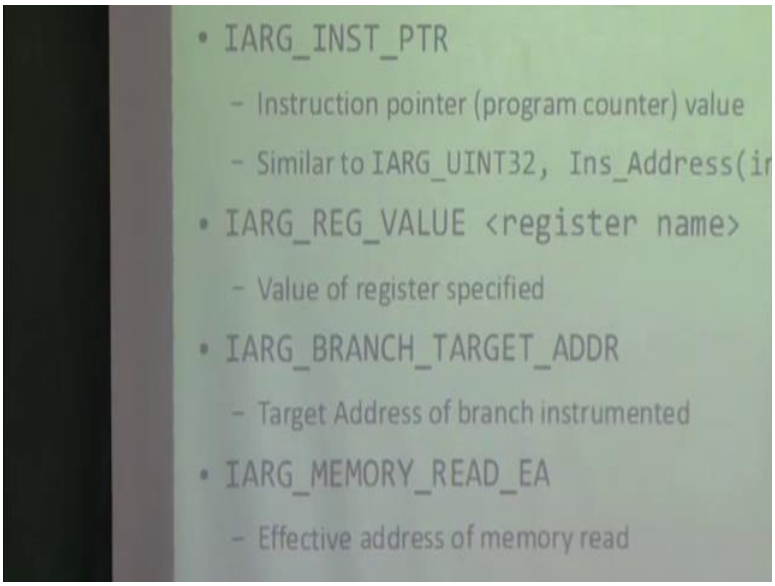
Student: For example, if I wanted to pass the file pointer instead of making, you can do that?

So, if I define file pointer over here, it would not be available to this i.

Student: i opening within instruction that is but and then pass it.

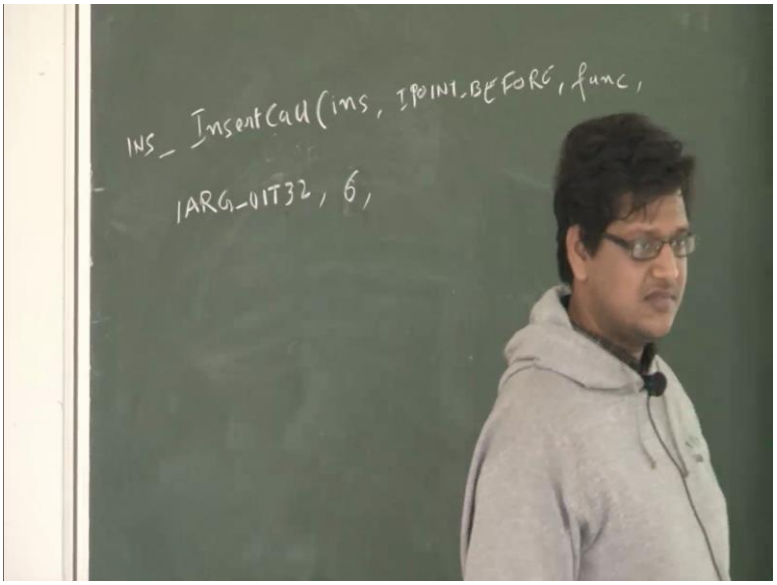
So, you can do that. So, is this...

(Refer Slide Time: 47:26)



So, example of arguments to analysis routine. So, we can pass a integer value. So, we can we could have written so.

(Refer Slide Time: 47:38)



I have not shown the INS-InsertCall, ins, IPOINT and function comma. So, this is what ins instrumentation? This is the ins instrument. So, these three arguments are still normally the same. If we want to pass a 32 bit integer value to this function as a input, we could say IARG u int 32. And then, we could some 6. So, what this is saying is that,

it is used like these. So, these macros some of these… So, this ((Refer Time: 48:39)) INT 32 macro this is not supplied around, just like IR.

It is supplying with a value. So, you have to give the value also over that. So, you give u int 32 and then the value after a comma, the value. So, it will pass the value 6 as a 32 bit integer value to the analysis function.

Student: It can be variable also. Need not be a constant.

It can be a variable also.

Student: And type cast, is not it.

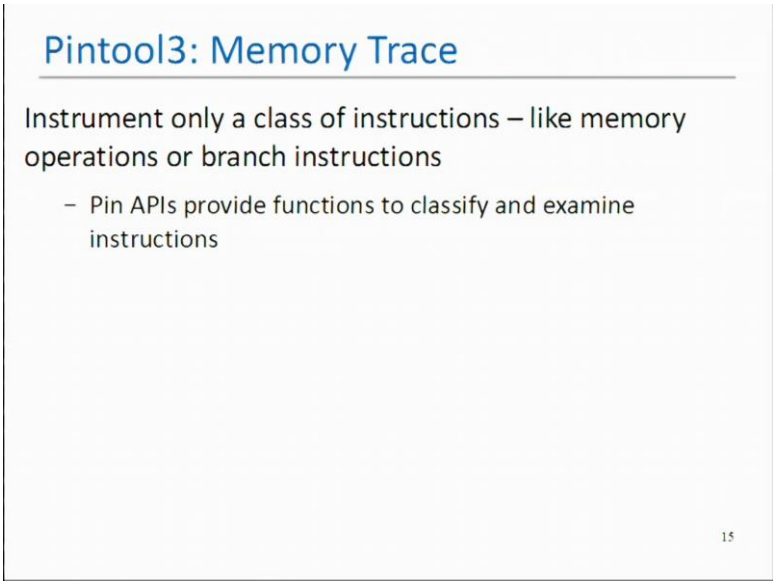
So, it is taken. So, this is just the type of this. And ((Refer Time: 49:13)) IARG instruction point is a macro which is actually. So, transformed into IARG_UNIT32 comma Ins_Address instruction and they are other so we can also pass register values to functions. I have not shown examples for this. But, IARG register value and then the register name. So, PIN would pass the value of that register. And we can pass the branch target address.

So, this is also a macro. It will get the branch instruction from the instruction of the BRANCH_TARGET_ADDRESS from the instruction. And it will pass it as a 32 bit. It will pass it as a instruction value to the function. And we can even pass memory address. We would see example, how to use them whether? And they are many more, you can pass any number of arguments.

Student: Where you could stop now?

There is one more example that I would explain now. So, till now we had been instrumenting, we have been inserting code at just adeque instruction. So, if you want to do a cash stimulation, we want to print the memory trace.

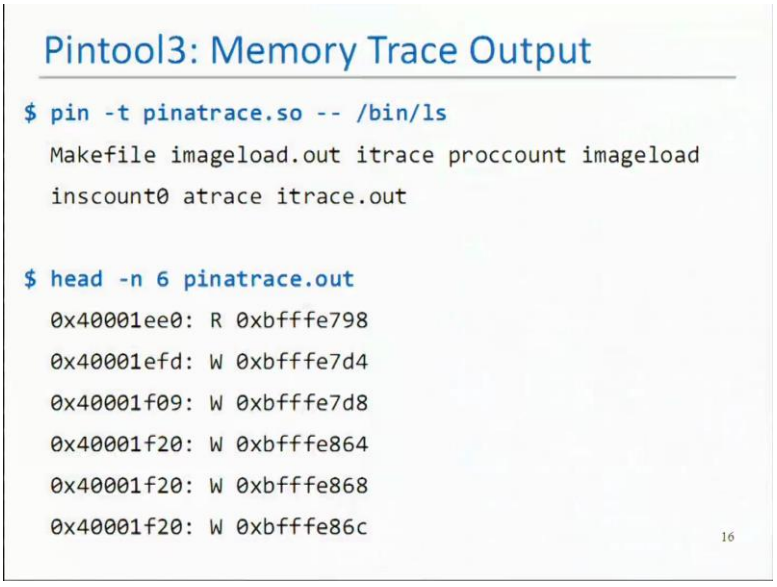
(Refer Slide Time: 50:25)



So, we want to insert code to print. Suppose that we are doing memory printing, a memory trace. So, we want to insert just before each load store instruction. Not before end instruction. We want to insert code specific to some instructions.

Student: So, instrument value, tell me what a memory trace is caused. Just a sequence of…

(Refer Slide Time: 50:55)



So, this is the memory trace. So, this is the instruction address. So, this instruction pc value and the memory trace is saying that, this instruction did a read from this memory

address. So, similarly this instruction did a write on this memory address. This is the memory trace. So, we want to collect this memory trace. And so till now we were instrumenting at each instruction, but if we can collect memory trace through that mechanism also, but that would be inefficient.

They are lot of almost one two third of the instructions are non memory instruction. So, we do not want some instrument. We want to instrument only the instructions, which are load store instructions. So, this tool pin trace does that, builds the memory trace. It is also there in the pin kit. And this is the code for it, a kind of pin kit.

Student: Do you want it to take it next time? So, I have put a link to include the source website. You can download and start playing with that. There are many, this is the website for pin. Link up with the course website also.

There is a manual there on the website. All these examples are taken from that manual. Kind of a tutorial, you should go through that tutorial. So, we will finish this up, this time.

Student: So, this is the conscience tutorial for you. Once we are done, we will post the password and all.