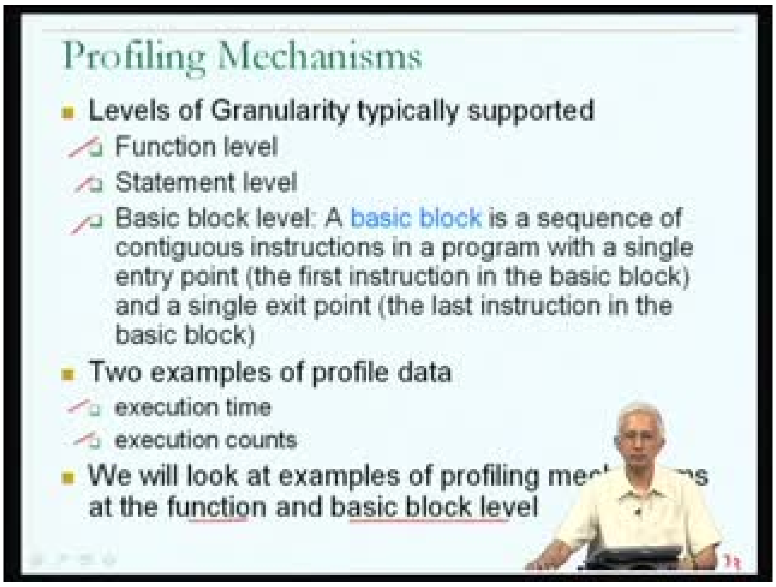
High Performance Computing

Prof. Matthew Jacob
Super Computer Education & Research Centre
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No.# 07
Lecture No. # 33

Welcome to lecture number 33 of high performance computing. In the previous lecture we had started to look at category of tools called profilers. The use of a profiler is to help us to identify the important parts of our programs,. So, that we can concentrate or optimization efforts on just those hopefully small, but important parts of our programs. And I had introduced the idea of profilers as automated mechanisms, through which the identification of such important parts of our programs can be made.

(Refer Slide Time: 00:58)



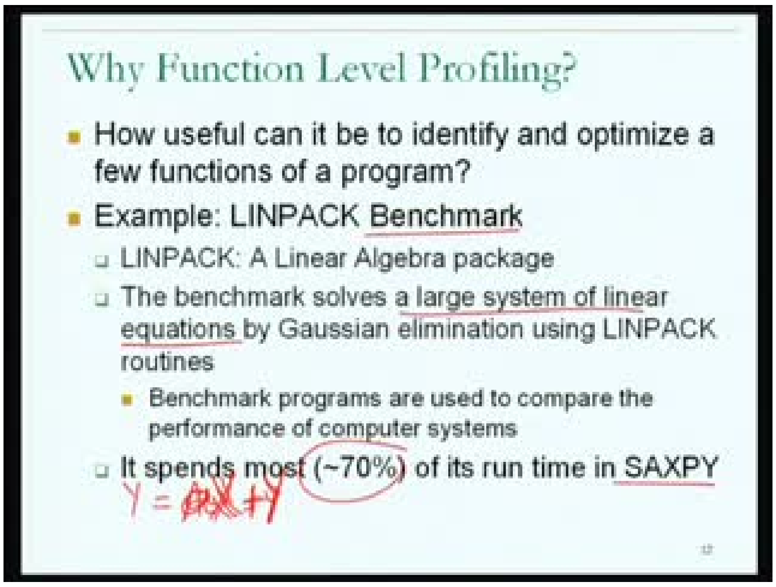
Profiling Mechanisms

- Levels of Granularity typically supported
  - Function level
  - Statement level
  - Basic block level: A basic block is a sequence of contiguous instructions in a program with a single entry point (the first instruction in the basic block) and a single exit point (the last instruction in the basic block)
- Two examples of profile data
  - execution time
  - execution counts
- We will look at examples of profiling mec[illegible]s at the function and basic block level

Now, we distinguish between different kinds of profilers in terms of the level at which they operate, which I had refer to in the previous class as the level of the granularity supported by the profiler. And I talked about the possibility of having function level profilers or statement level profilers or basic block level profilers, and a function level profiler would help us to identify the important functions of our program.

A basic block level profiler would help us identify the important basic blocks of our program. In addition, profilers might differ in whether they are giving us breakups of execution time or just giving us other pieces of feedback about the performance of our program, such as the number of times that a function was executed or a basic block was executed. And we are going to look at two mechanisms one at the function level and one at the basic block level.

(Refer Slide Time: 01:36)



Why Function Level Profiling?

- How useful can it be to identify and optimize a few functions of a program?
- Example: LINPACK Benchmark
  - LINPACK: A Linear Algebra package
  - The benchmark solves a large system of linear equations by Gaussian elimination using LINPACK routines
    - Benchmark programs are used to compare the performance of computer systems
  - It spends most (~70%) of its run time in SAXPY

$Y = aX + Y$

Now, why would function level profiling be a good idea? Now, this question is essentially rephrased as how useful can it be to identify in optimize a few functions of a program? Many of you would have written programs which are extremely large, typically the programs that you are interested in optimizing, a complicated otherwise they would be little reason to try to optimize a program, and the more complicated a program is a larger it is likely to be, and if the program is large and has 100s of functions. How could it possibly help to just optimize 2 or 3 functions of that program?

Now, just to give you some feel for what might be a typical property of large programs, I am going to give you the example of real world program called the LINPACK Benchmark. Its referred to as the benchmark, because it is used in doing performance comparisons of pieces of hardware. The idea of a benchmark program is that one where runs the benchmark program on a piece of hardware measures how much time the program takes, and then if you run the same program on another piece of hardware and it takes less time. You will know that the second piece of hardware is faster or better and one could use analysis based on the performance of benchmark programs, in making decisions about what computer to buy etcetera.

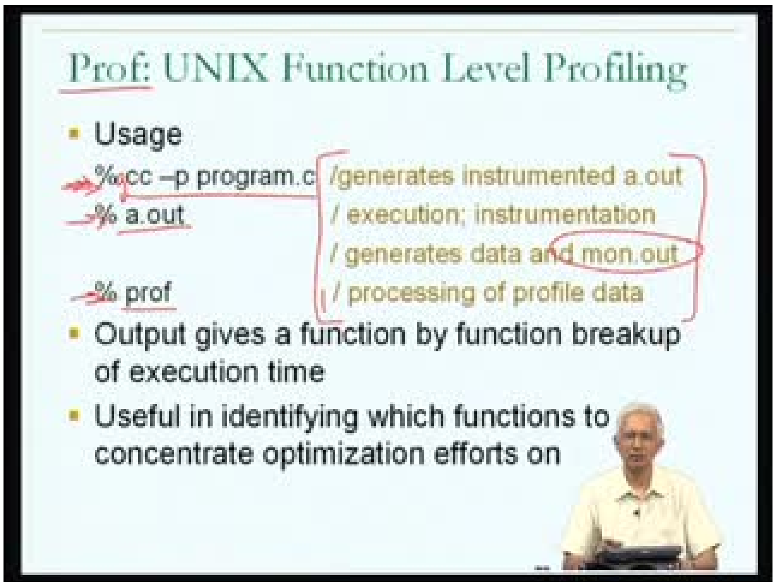
So, the whole idea of benchmarking is interesting, but the LINPACK Benchmark is one example of a benchmark. And LINPACK itself is acronym for the name of a Linear Algebra package, what we mean by linear algebra package is the package that does library of functions for computation on vectors and matrices multidimensional matrices. Now, the LINPACK Benchmark itself uses the LINPACK library to solve a large system of linear equations using Gaussian elimination. So, it uses many of the functions inside the linear algebra package to solve a particular problem, which is what I am referring to you solving a large system of linear equations.

Now, the LINPACK Benchmark is a fairly large program in terms of if you look at the size of the program as well as the size of the library it is a fairly large, includes many functions and the execution time of the LINPACK Benchmark is fairly substantial. So, it is in fact a commercial type of a program in that is used to make by some people even in making purchase decisions, or comparing the performance of large computer systems.

Now, as it happens if one analyzes the LINPACK Benchmark in terms of its function level profile strangely enough one would find that a significant percentage of the execution time of the LINPACK Benchmark is spent in one function. In fact, about 70 percent if I remember right of the execution time of the LINPACK Benchmark is spent in a function called SAXPY. And we have come across the name SAXPY before we learned about its sibling DAXPY. Do you remember that DAXPY was double procession. A X plus Y we saw this not too long ago, and SAXPY is a single procession version of A X plus Y. It is multiplying the vector X by a scalar A and adding this two the vector Y, and making this a new value of the vector Y. A simple operation which we represented by a very small for loop. In the case of or see example not in a few lectures back, 70 percent of the execution time of this real world benchmark is spent in one function and the size of the function is just one or two lines.

So, very clearly this motivates the idea of function level profiling. It may be the case that is by just concentrating on a small number 1, 2, 3 maybe 4 functions. You may be addressing a large percentage of the execution time of the program that you are trying to optimize.

(Refer Slide Time: 05:30)



Prof: UNIX Function Level Profiling

- Usage

```
%cc –p program.c   /generates instrumented a.out
% a.out            / execution; instrumentation
                   / generates data and mon.out
% prof             / processing of profile data
```

- Output gives a function by function breakup of execution time
- Useful in identifying which functions to concentrate optimization efforts on

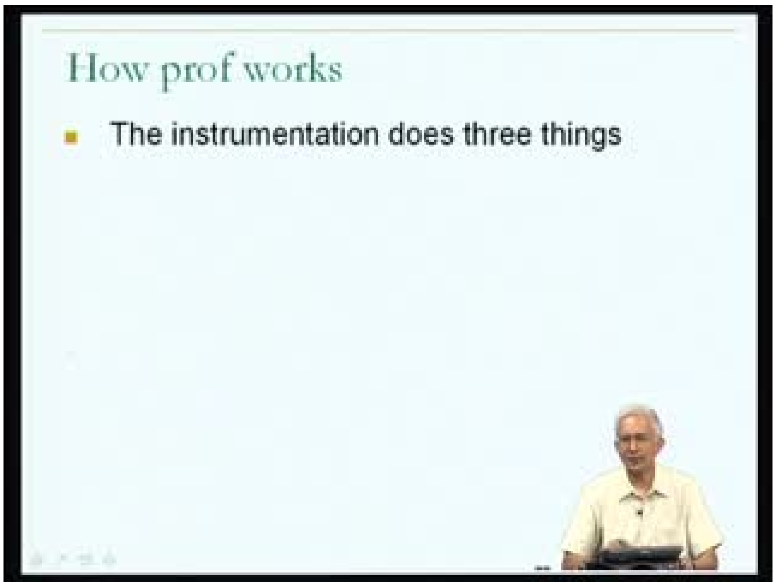
So, function level profiling may not be a bad idea. Now, the function level profiling mechanism that we are going to talk about is known as prof. p r o f, prof standing for profiler and it is the UNIX function level profiling mechanism. The way that you use prof and again I am showing you the interaction in terms of shell interaction with the shell starts these comments; obviously, not part of the shell I have just added them for explanation of what is happening.

So, you start if you want to use the prof profiling mechanism then you must compile your program using the minus p option. So, g c c or c c minus p program dot c and what this does is it causes a special version of the executable to be created. So, rather than just compiling your program dot c to generate an a dot out file.

g c c will now compile your program program, dot c it include within the program the a dot out instrumentation in other words instructions, through which when the program is executed profile data is gathered and subsequently. If you run a program called prof you can see the profile data in a human readable form. So, it is a 3 step process you start by asking g c c to compile your program. So, that it is an instrumented executable a dot out not the ordinary a dot out subsequently when you run that a dot out a file full of profile data gets return out, and you can subsequently use another program to look at the contents of that profile data in human readable form on the screen.

What is the output that you see when you run prof the output that you get is function by function a breakup of the execution time the fraction the percentage of execution time that was spent in that particular function. In addition it also gives you the number of times that each function was executed. These two pieces of information will be useful in identifying which functions you should concentrate your optimization efforts. For example, if you had done this on the LINPACK Benchmark, you might find out that 70 percent of the execution time was spent in SAXPY, which was called a huge number of times.

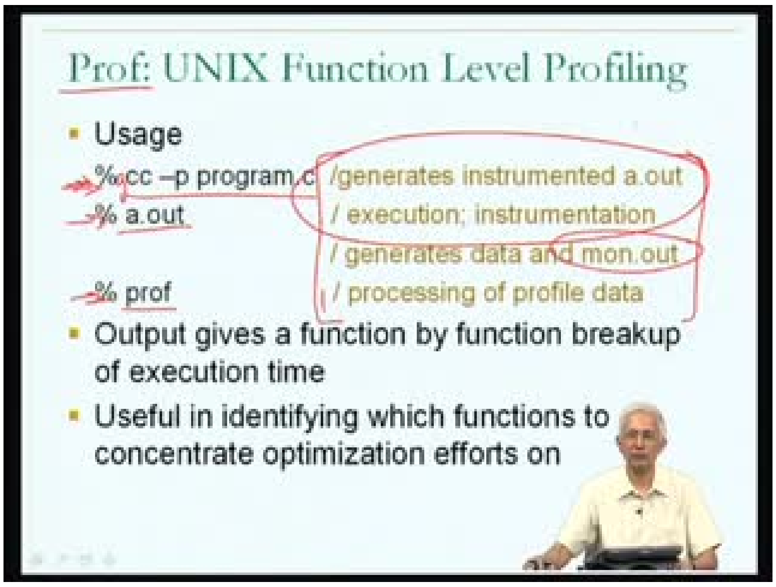
(Refer Slide Time: 07:42)



How prof works

- The instrumentation does three things

(Refer Slide Time: 07:51)



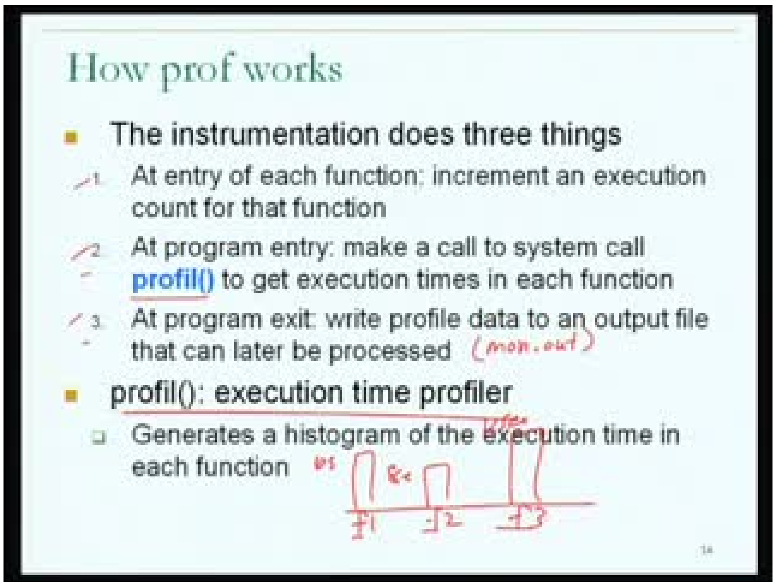
Prof: UNIX Function Level Profiling

- Usage

```
%cc –p program.c  /generates instrumented a.out
% a.out           / execution; instrumentation
                  / generates data and mon.out
% prof            / processing of profile data
```

- Output gives a function by function breakup of execution time
- Useful in identifying which functions to concentrate optimization efforts on

(Refer Slide Time: 07:56)



How prof works

- The instrumentation does three things
  1. At entry of each function: increment an execution count for that function
  2. At program entry: make a call to system call profil() to get execution times in each function
  3. At program exit: write profile data to an output file that can later be processed (mon.out)
- profil(): execution time profiler
  - Generates a histogram of the execution time in each function

Now, first question how does prof work? How does this entire mechanism work? Now, it turns out that the instrumentation which is included within the a dot out file that will be executed when you run the program the instrumentation does three things. One thing that it does is at the entry of each function. In other words, at the beginning of each function in your program. It includes an execution count incrementing instruction.

Therefore, what prof what the instrumentation does is, it maintains a count of the number of times each instruction, each function is entered. How does it do? This it will actually has a set of counters, one for each function and at the beginning of each function have when the function is entered it increments the count for that particular function, then when the program terminates it prints out all these counters into the mon dot out file. So, one piece of instrumentation that is done is that at the entry of each function a counter for that function is incremented. So, that the number of times of that function was executed can be kept track off.

Now, another thing that the instrumentation does is that at the entry point of the program. In other words, in the early parts of the program it makes a call to the system call called profil p r o f i l. And what profil is going to do, is to get estimates of the execution times within each function. Therefore, the instrumentation itself does not do anything with timing mechanisms all that the instrumentation does this system call which is known as profil.

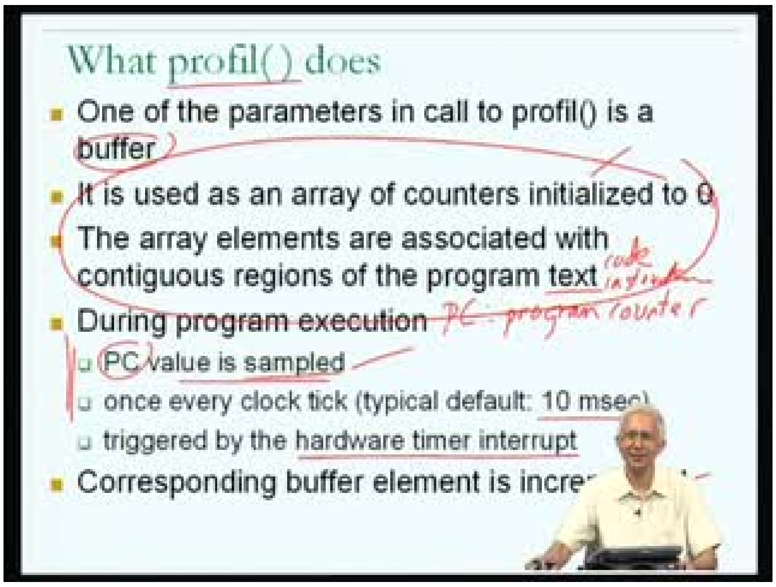
Finally, at the third thing that the instrumentation does at the exit point of the program. In other words, just before the program terminates. At that part of the program it causes the contents of the various counters that have been accumulated to be written into the output file which we understand has a default name of mon dot out. So, the subsequently the output file when the program is run the output file will contain the information that can be used to understand what happen when the program run.

So, the instrumentation does these three things: very simple incrementing a counter at the beginning of each function; calling profile at the beginning of the program; and dumping the various counters of information that has been maintained at the exit point of the program. So, instructions to do these things are included by g c c into the a dot out that is what we mean by instrumentation. So, now we have reduced our problem to trying to understand what does the profil system call do. If you look at the profil system call details you will find out that profile is described as an execution time profiler. So, it was included it is included as a system call exactly this purpose to provide function level profiles of execution time.

And essentially what it does is it generates a histogram of the execution times spent in each function. What I mean by histogram is for each function there may be function 1, function 2, function 3. The histogram, if the details of how much time was spent in that particular function. So, this could be 10 seconds, this could be 8 seconds, this could be

12 seconds and so on. So, essentially that is what profile does. It generates a histogram for each function, exact how much time was spent in that particular function as far as the execution time of the program is concerned.

(Refer Slide Time: 11:07)



What profil() does

- One of the parameters in call to profil() is a buffer
- It is used as an array of counters initialized to 0
- The array elements are associated with contiguous regions of the program text code instr[illegible]
- During program execution PC - program counter
  - PC value is sampled
  - once every clock tick (typical default: 10 msec)
  - triggered by the hardware timer interrupt
- Corresponding buffer element is incre

Let us just see in more detail how profil does this. Now I am referring to profil, remember that this is the name of the system call which is used by the prof mechanism. Now to understand this quickly, profile is a system call. It has many parameters. One of the parameters in a call to profile is a buffer or an array, and this array is used as an array of counters each counter being initialized to zero.

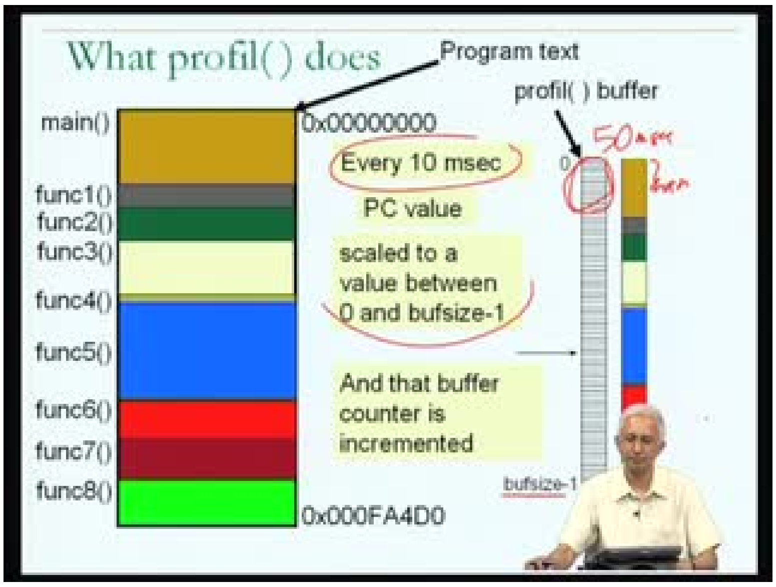
Now, the different counters inside this buffer array are each associated with some contiguous region of the program text. You will remember that when we refer to the text of a program we are referring to the code or the instructions of the program. So, what is this means? This means that the profil has a buffer an array of counters. The counters are all initialized to 0, and each of the counters is associated with some region of the program of the instructions of the program

Now, during program execution when the program is actually executing every once in a while the program counter value is sampled. Remember that PC was a name of one of the special purpose registers inside the CPU, and at any given point in time that particular register contains the address of the instruction which is currently being executed. Therefore, and what I mean by sampled is that every once in a while the system looks at the current value inside the program counter register. In other words, what it means to sample the program counter every once in a while, is that every once in a while due to the profil system call it keeps track of which instruction is currently being executed. And this is done with some period as I said it is done every once in a while. In many systems this is done once every 10 milliseconds.

And this activity would actually be triggered by that hardware timer interrupt. You will remember that within one of the pieces of hardware in any computer system is a piece of hardware that periodically generates an interrupt. And this is critical to the functioning of the operating system. For example, in making decisions about doing context which is and so on. So, one other piece of functionality that would be included in the hardware timer interrupt handler we now see its going to be related to profiling.

Possibly the sampling of the program counter would be initiated out of the hardware timer interrupt handler. If that is so, required by the program what then happens is depending on what program counter of value we has been identified as a current program counter value the corresponding buffer element counter is incremented. So, if we had found out that a particular load word instruction which has address hex A000 is the one that is currently being executed, then the particular counter in the buffer relating to that particular instruction address would be incremented. And this is how the profil system call gets an idea about the different instructions of the amount of execution time that is spent in each part of the program. Let us just look at this a little bit more carefully.

(Refer Slide Time: 14:13)



Every 10 msec

PC value

scaled to a value between 0 and bufsize-1

And that buffer counter is incremented

So, what profil does is I am just recreating what happened, what was described in the previous slide. So, let us suppose this is our program text, this is the different instructions making which my program is made up. What I am showing on the screen the program contains many functions. So, I am assuming that I have functions called main func1 func2 up to func8. So, this particular program we saying as 9 functions, and exactly where each program begins and ends. Because I wrote the main function and the compiler compiles the main function into many instructions, and then the compiler compiles the func1instruction function into many instructions and so on.

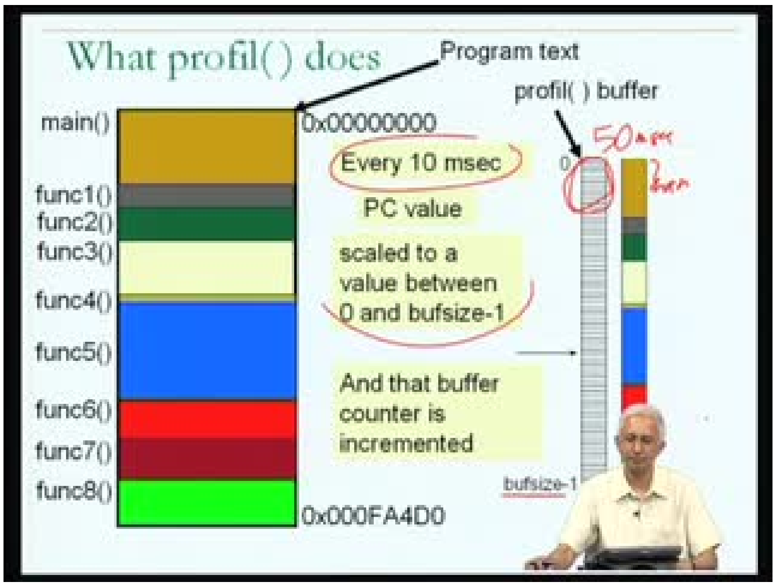
So, when I look at the program text I can actually view the program text as the different functions, and here I have drawn each of the 9 functions and shaded them in a different color. Now, you should also bear in mind that in looking of the program text I could also view each of the functions in terms of its start address and its end address, because we can count how many instructions there are as part of each function. And then we can associate the start address and the end address of each function.

Now, at the other side profil is using as I mentioned a buffer that is an array of counters, and there are going to be some number of counters in that buffer the number of counters in the buffer need not be the same as the number of functions in the program. I need not be the same as the number of instructions in the program it just has some number of counters within the buffer.

Now, once every 10 milliseconds remember the program counter is sampled. So, let us suppose that it in one particular sample now the thing to bear in mind is the size of the buffer. Once every 10 milliseconds a particular program counter value is obtained and let us suppose that is some particular program counter value. Now the size of these buffer is going to be decided based on the system call, it is going to be some size. I will just refer to it as bufsize. So, it has elements counters going from counter 0 up to counter bufsize minus 1.

Now, whenever a PC value is sampled that PC value can be scaled to a value between 0 and bufsize minus 1, depending on what the value is. For example, if the value is 0 we might scale it to 0; if the value is the largest possible value we might scale it to largest possible buffer size value; if its half way in between we might scale it to the halfway in between buf counter counter value.

(Refer Slide Time: 14:13)

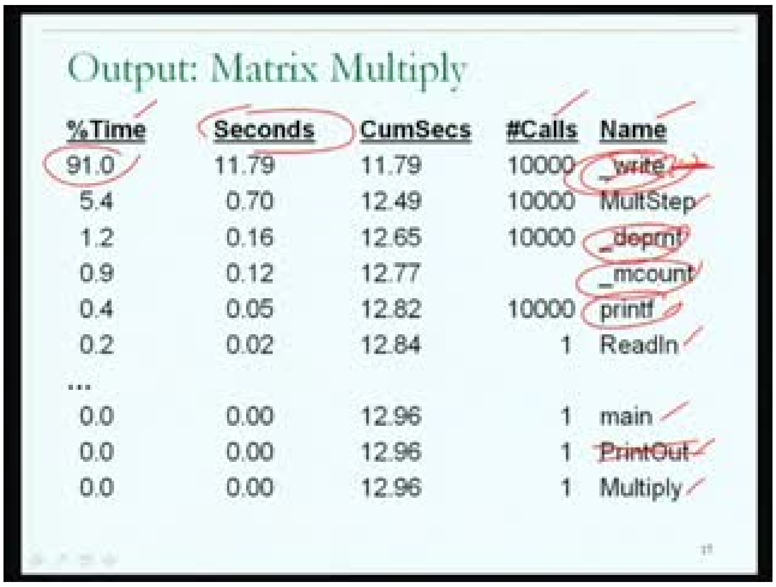


So, it some scaling is done of the PC value to a value between 0 and bufsize minus 1, and this is the value that is currently corresponds to that P C value then that buffer counter is incremented. So, that is how the PC sampling mechanism happens. So, as slowly as the program executes thanks to the profil system call once every 10 milliseconds, one or the other of these buffer counter of these counters gets incremented.

Now, how do we relate to this execution time of the functions in the program that does not seem to be any relationship between the buffer elements or the counters inside this buffer and the functions of the program. However we can associate a range of buffers a range of counters with each of the functions in the program, possibly based on the number of instructions in that particular function.

So, for example, if main occupies 20 percent of the instructions in the program then I know that the first 20 percent of the buffer counters are going to be associated with main, and therefore there is this relationship that we can make between the counters back to the functions. And therefore, all that the prof output has to tell us or has to be manipulated to do is all of the counters which relate to the region of program text that were mapped into by the scaling function. Relating to a particular function can be added together to given estimate of the amount of time that was spent in that particular function.

(Refer Slide Time: 18:45)



Output: Matrix Multiply

%Time	Seconds	CumSecs	#Calls	Name
91.0	11.79	11.79	10000	_write
5.4	0.70	12.49	10000	MultStep
1.2	0.16	12.65	10000	_doprnt
0.9	0.12	12.77		_mcount
0.4	0.05	12.82	10000	printf
0.2	0.02	12.84	1	ReadIn
...				
0.0	0.00	12.96	1	main
0.0	0.00	12.96	1	~~PrintOut~~
0.0	0.00	12.96	1	Multiply

How much time should be associated with each of the counts you will remember that we were doing a sample once every 10 milliseconds. And therefore, if the total of all the counts within the region associated with main was five, then that could be associated to have a value of 50 milliseconds, and similar calculations could be done for all of the functions. This is essentially what is happening behind the scene as far as the profile system call is concerned. The ultimately the output would look something like this over here I am showing you the output generated by running prof on a matrix multiplication program.

Now, in this particular program I actually had a main function there was a function to read the matrices L in there was another function to print the result out. There was a function to do the multiplication of the matrices within the function to print the matrices out there was a call to print f which is a function to do printing within the function to do the multiplication. There was a sub a smaller function which was called which does a step of the multiplication and then the additional functions which you see over here are actually functions which are called out of printf or out of the other functions in the program. They are actually not functions which are written by me when I wrote the program, but are function which were added due to the fact that they were possibly part of the libraries that my program called.

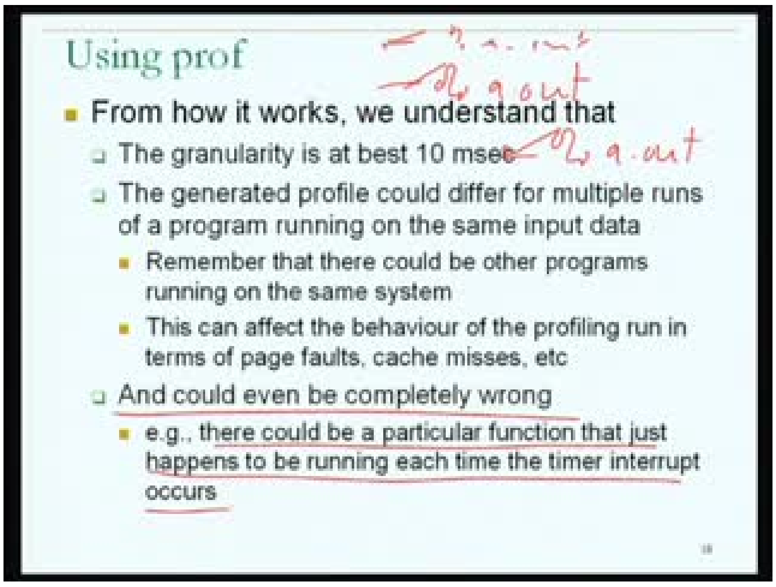
Now, if I look at the output of prof I notice that there was one entry for each function, and I am told exactly how many times that function was called. I am also given an estimate of the percentage of time of the total execution time of this program that was spent within the function. So, for example, I am told it 90 percent 91 percent of the execution time was spent in the right system call was spent in a function called write, and I am also given some information about the number of seconds that that percentage was amounted to.

Now, you may be wondering how come this exercise seems to be telling us nothing useful in that 90 percent of the execution time was spent in a function that is part of the library. And that I have no control over now in this particular case the size of the problem was fairly small I was multiplying a relatively small matrix.

So, that the execution time of the program was dominated by the activity of writing the result out on the screen. And that is a slightly artificial situation it is probably more, it would have been more beneficial to me if I was interested in optimizing the program to actually leave out all the functions, which have to do with in essentials such as printing the output on to the screen. But I am trying to optimize the program I want to optimize the parts of the program that do the calculation. And therefore, it would probably have been a better idea for me to have done the profile run on a version of the program which does not do the output at all as we clearly see the output was dominated or the profile output was dominated by the input and output functionalities of this program.

So, do we have any other guidelines that could be used in deciding how to use prof? One very clearly was do not just use prof if your objective is to use prof in order to find out which functions you should spend your optimization efforts on. Then certainly do not include all the calls, to input or particularly output types of functionalities in your program unless they are essential for the functionality of the program.

(Refer Slide Time: 21:46)



Using prof

- From how it works, we understand that
  - The granularity is at best 10 msec
  - The generated profile could differ for multiple runs of a program running on the same input data
    - Remember that there could be other programs running on the same system
    - This can affect the behaviour of the profiling run in terms of page faults, cache misses, etc
  - And could even be completely wrong
    - e.g., there could be a particular function that just happens to be running each time the timer interrupt occurs

So, what else can we learn about how to use prof? From our understanding of how it works, now a few things that we can understand first of all is that prof operates by using profile, the system call profile. And what the system call profile does is that it causes the PC value the program counter value to be sampled once every 10 milliseconds, and subsequently whichever function happens to be executing at that point in time will be given a weightage of 10 milliseconds. The net effect is that the granularity of this profiling mechanism can at best be viewed as being 10 milliseconds. It is not capable of giving me a finer picture of how the execution time is being divided across the different functions of the program.

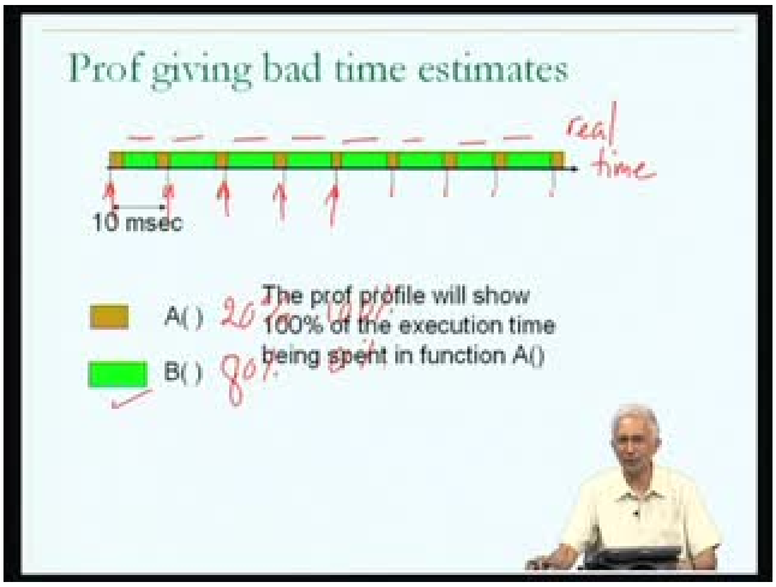
Another thing which I would note from the description of how prof works is that the profile that is generated could differ for multiple runs of a program, even if I am running the program on the same data in each of those runs. And the question you’ll ask is how could this happen? Now, the suggestion is that I could run a dot out and then I could run a dot out again, and I could run a dot out again. And each time that I did this I could get a different profile and that the question is how could this happen? And the answer that as you can realize it is at the time that you do these profiling experiments they could be other programs running on the system, and the other programs will be using the different sharing the different resources of the computer system with your program.

Which is currently going through the profile run therefore, the behavior of the other program could affect things that effect your program such as the contents of the physical memory, the contents of the caches, and therefore, your program might see different numbers of page faults or different number of cache misses, due to the fact that there are other programs running on the system. And therefore, each time that you do the profiling experiment you could get different results.

So, this is something that we have to bear in mind from our understanding of how the prof mechanism works lastly even though we have not gone into this. As yet I will go into this shortly the output that you get from prof could actually be completely wrong, and this is largely going to be an artifact of the fact that it samples the program counter once every 10 milliseconds and this is scary. So, we do need to understand how this could happen and if. So, what could we could do about it.

Now, the general idea is that in the worse case every time that the program counter is sampled it could be that there is a particular function that just happens to be running at that point in time, and let me just illustrate how this could happen.

(Refer Slide Time: 24:34)



Prof giving bad time estimates

The prof profile will show 100% of the execution time being spent in function A()

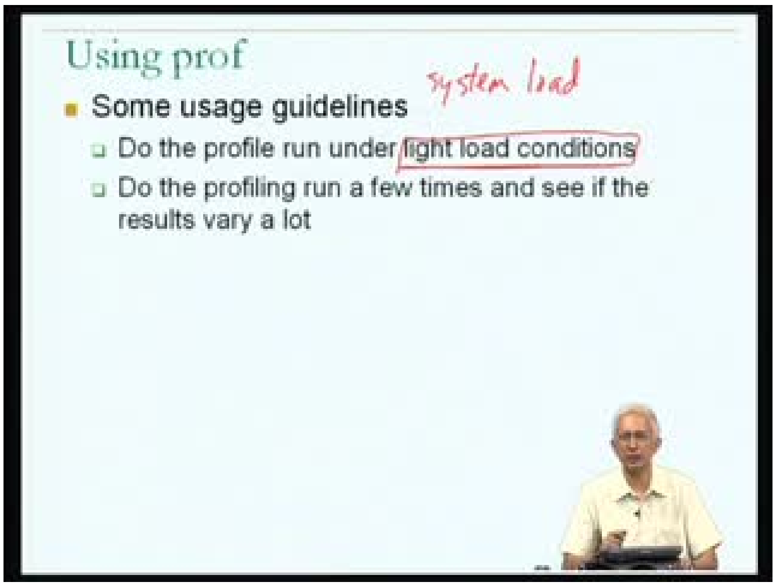
Now, this is an example of how prof could give really bad time estimates or a really bad inaccurate profile of the breakup of execution time across functions. Now, you will remember that we are talking about the real timeline, because we are talking about something that happens triggered by once every 10 milliseconds by a hardware timer interrupt. Therefore, the 10 milliseconds that we talked about was not virtual time of process p1 or virtual time of process p2 what was along the real timeline the elapse timeline which once every 10 milliseconds.

Therefore once every 10 millisecond, the elapse timeline this is real time or elapse time has to be viewed as being broken up into these chunks of size 10 milliseconds. And let me assume for the movement that there is only one program in execution, forget about the interference from other process is.

So, it could happen that the first my program starts executing it starts by executing a function. And I am designating that function by the brown time on the real timeline. So, that could be the function which is function A of my program. So, function A runs for a few milliseconds or less, and then it calls function B which is the green function, and then function B calls function A again, and then after little while of execution function A calls function B, and so on.

So, you can see that what is happening is that every time the 10 millisecond timer interrupt happens it just happens that function A happens to be running. But the bulk of the time in between it turns out that function B is running. Therefore, realistically if you look at this diagram you might say function B is running for about 70 percent of the time; and function A is running for about or may be 80 percent of the time; and function A is running for about 20 percent of the time. And therefore, if I had to optimize one of these functions I would try to optimize function B, but what is the output from prof going to be as far as this particular program is concerned? Unfortunately every time the timer interrupt happens function A is running. And therefore, the estimate the prof is going to give you is that 100 percentage of the time is spent in function A, and 0 percentage of the time is spent in function B.

(Refer Slide Time: 27:06)



Using prof

- Some usage guidelines
  - Do the profile run under light load conditions
  - Do the profiling run a few times and see if the results vary a lot

system load

So, the prof profile will show 100 percent of the execution time being spent in function A, and this could happen may not be frequently occurring situation. But it something that could happen, in other words prof could be completely wrong and we have to somehow take this into account. Otherwise we could be in deep trouble. Therefore, we must have some kind of guidelines for using prof, based on our understanding of what is happening behind the scenes when prof is used. And here some possible usage guidelines.

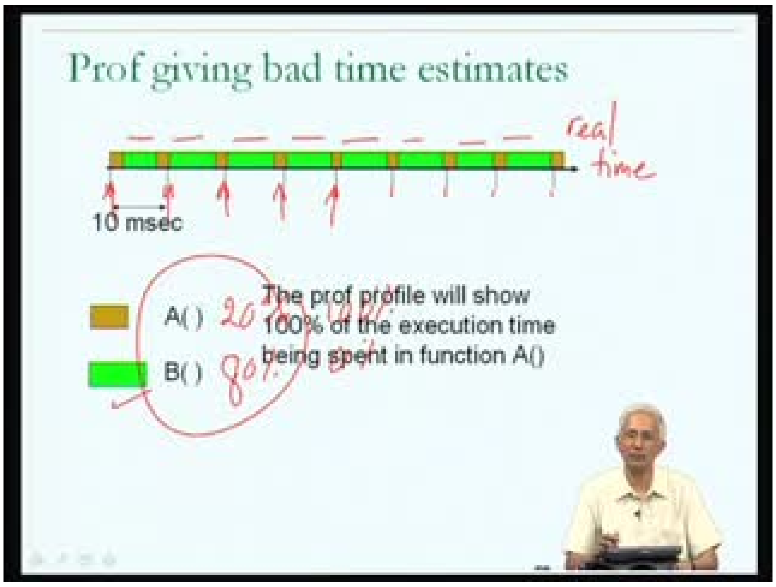
Now, that the first usage guideline which people very often make is that you need to make sure that you do the profile run under as light load conditions as you can, and what do we mean by light load conditions. We are talking here about the number of programs running on the system, people often referred to the load on a system by the term system load, and you could quantify the load on a computer system by the number of programs or the number of processes that are currently active on that computer system.

So, to talk about light load conditions means that we are talking about conditions where the number of process is the number of programs currently running on that computer. system is as is minimum or is as low as possible, what is the reason for using this as a prof usage guideline we saw that if there are many programs running on a system then I have the problem, that if I run my program a dot out then it will suffer from the cache misses page faults, which are artificially created because the other programs are used sharing the resources on the computer system.

Therefore, if I wanted to find out the true properties of my program I would actually want to study my program under conditions where there are few interactions or few nuisances created by the other programs in execution on the system and that is why the recommendation is to use profile run under light load conditions.

Now, even if you do a profile run under light load conditions there is a possibility that when you run it 2 or 3 times you get different results. In one you may be told that 20 percent of the execution time is in program A; in the other you may be told that 33 percent of the execution time is in function A.

(Refer Slide Time: 29:00)

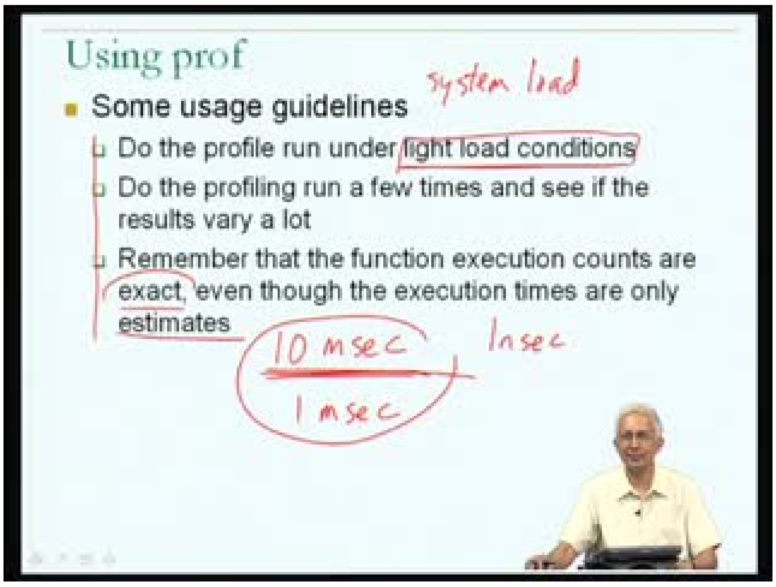


Prof giving bad time estimates

The prof profile will show 100% of the execution time being spent in function A()

Therefore, generally it make sense not just to do the profiling run once, but run it a few times and see if the results vary substantially. For example, if we actually had a situation like what we see over here, but then we ran the program the profile run 2 or 3 times. Then there is every chance to believe that for some of the runs this exact skewing, this exact match between the time that the time limit up occurs and a happen to be executing may not be the case.

(Refer Slide Time: 29:26)



Using prof

- Some usage guidelines
  - Do the profile run under light load conditions
  - Do the profiling run a few times and see if the results vary a lot
  - Remember that the function execution counts are exact, even though the execution times are only estimates

system load
10 msec
1 msec
1nsec

So, if I ran if I did 10 profile runs I may find out that 3 of them give me the wrong profile and the remaining 7 give me something else, may be closer to the accurate profile. Therefore, to overcome the problem with being completely wrong it might make sense to do the profiling run a few times. And if the results vary quite a lot then do it a few more times, and then assume that the majority output from the many runs that you did is the one that is likely to be more typical of the behavior of your program.

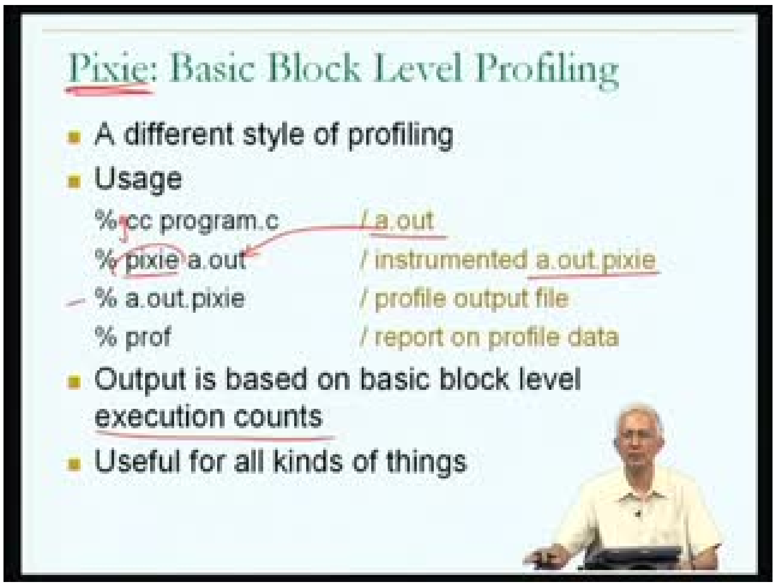
Now, another piece of information that should be born in mind is remember that prof in addition to giving you this estimate of the percentage or the amount of time that was spent in each function, also gives you the number of times that each function was executed; and that count of number of times that each function was executed was generated by incrementing a counter that was incremented within the early parts of the function. And therefore, the count of the number of times that each function was executed is exact the timing estimates are the break of execution time are only estimates and they could be bad estimates, but the count of the number times each function was executed will be exact. And therefore, one can always use execution counts and are understand that they are precise and that there is no reason to think that they could be wrong in any way.

Now, with this so, we have an idea about one level of profiling at a function level and we see that there are certain problems with using the function level, but with appropriate guidelines we should be able to get useful information out of using a mechanism like prof. Now, you may be wondering about that 10 millisecond interval of sampling the program counter on a scale where we have processers that are faster than 1 gigahertz. You will note that in 10 milliseconds there will be a huge number of instructions that could be executed, if we have a processor which is at 1 gigahertz than every nanosecond it might be executing a single instruction; and therefore, in 10 milliseconds you can see that there are millions and millions instructions they get executed therefore, is sampling the program counter once every 10 milliseconds adequate should not they be sampling at more frequently.

Now, you will see that there are some processors some systems in which the sample the program counter once every millisecond, but that is pretty much the far this set you may find it turns out that it is adequate for the purpose, that we are talking about to sample the program counter relatively in frequently, because for a long running program you will get a large number of samples and statistically as long as the number of samples that one gets is adequate to give us an idea of the distribution of time across the different functions. Then the number of samples the interval between two samples is not that important a consideration therefore, there is not too much of the concerned about the fact that the sampling happens at 10 milliseconds or at best 1 milliseconds.

Now, with this let us move on to the second profiling mechanism that I would talk about and this is an example of a profiling mechanism which happens that the basic block level.

(Refer Slide Time: 32:11)



Pixie: Basic Block Level Profiling

- A different style of profiling
- Usage

```
% cc program.c          / a.out
% pixie a.out           / instrumented a.out.pixie
% a.out.pixie           / profile output file
% prof                  / report on profile data
```

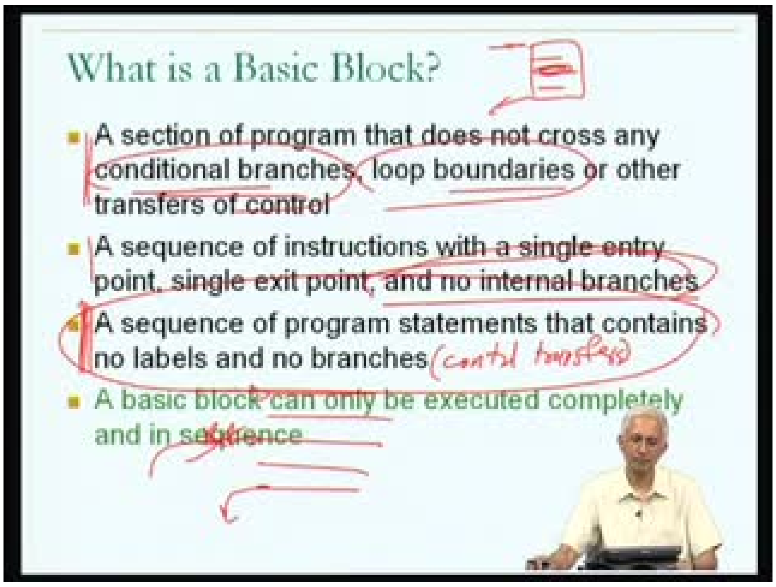
- Output is based on basic block level execution counts
- Useful for all kinds of things

And I am using one of the early names one of the early tools which used basic block level profiling which appeared sometime in the 1980s and was known as pixie. So, this is going to be as you can clearly imagine and completely different style of profiling from what was happening in the case of prof, and I will suggest a mechanism which could be used for usage this is the usage mechanism from the original pixie tool. So, the idea of here is the once again you compile your program, but without any particular request to the compiler to do any instrumentation.

So, this will cause an a dot out file to be executed or to be generated subsequently you use a program which does the instrumentation for you this was the mechanism that was done in at the time of pixie. So, a program called pixie which would take in the a dot out and generate an instrumented executable which it called a dot out dot pixie to the instrumentation would be done not by the compiler, but by another program subsequently you would run the instrumented version of the program a dot out dot pixie and this would generate an output file which you could examine using a program like prof.

Now, unlike the function level profiling here the output is going to be based on properties at basic block level, and there is going to be no attempt to estimate the amount of time spent in executing a basic block. But rather what the pixie mechanism gave you was execution counts for the different basic blocks of the program. So, it is basically an execution counting mechanism; a counting mechanism not a time estimation mechanism, and this is actually was useful for all kinds of different purposes in addition to identifying the important parts of programs it was also used by computer architects for various kinds of architectural studies of programs.

(Refer Slide Time: 34:03)



What is a Basic Block?

- A section of program that does not cross any conditional branches, loop boundaries or other transfers of control
- A sequence of instructions with a single entry point, single exit point, and no internal branches
- A sequence of program statements that contains no labels and no branches (control transfers)
- A basic block can only be executed completely and in sequence

Now, I have given you in the previous class, idea about what a basic block is I defined it as sequence of contiguous instruction in a program such that the only entry into that sequence of the instructions is that the first instruction in the program is the first instruction in the basic block and the only exit from the basic block is the last instruction in the basic block.

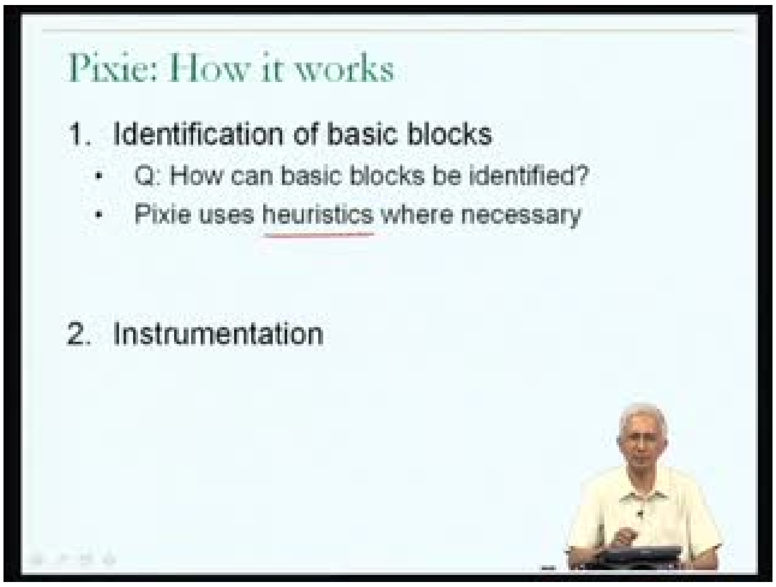
So, in general one could infer from this that any one of these three could be viewed as being interpretations of what a basic block is in the first. I described a basic block as a section of a program that does not cross any conditional branch loop boundaries or other transfers of control. And you can sort of see that this is in sync with the first explanation of what a basic block was because if there is a basic block which the only entry point is the first instruction, and the only exit point is the last instruction. And there only instructions in between then quite definitely they cannot be any branch into the middle of the basic block, which means that it cannot cross any conditional branches etcetera.

So, one could argue that this is consistent with our first explanation similarly it could be described as a sequence of instructions with the single entry point, single exit point and no internal branches. So, this no internal branches is an additional observation you notice that from our first explanation, I had said that you could only enter a basic block from the first instruction, and you could only leave by the last instruction, but if there was a branch within the basic block then you could have left from within the basic block. And therefore, they can clearly be no internal branches. So, this also consisting with our previous explanation and finally, this relationship between our first explanation of the basic block and our current two explanations of the basic block. You notice that the current two explanations of the basic block are describing basic blocks in terms of control transfer instructions conditional branches loop boundaries and so on.

And therefore, one could actually sort of infer that the basic block has something to do with control transfer instructions, and one can view it as a sequence of program in statements or instructions that contains no labels and no branches no control transfer instructions. That is why I mean by branches and what does it mean to be a label? A label is the target of a control transfer instruction. So, this is somewhat more functional description of what a basic block is. It is some sequence of instructions in a program which have the property that you cannot branch into in other words there is no instruction in the sequence which has a label, and further no instruction in this sequence is the control transfer instruction.

And from this it follows that you could only have entered from the top, and you could only leave from the bottom which was our starting definition. So, this gives us the functional mechanism through which we can identify basic blocks and programs, and it is a better way to look at programs before once to understand basic blocks.

(Refer Slide Time: 36:53)



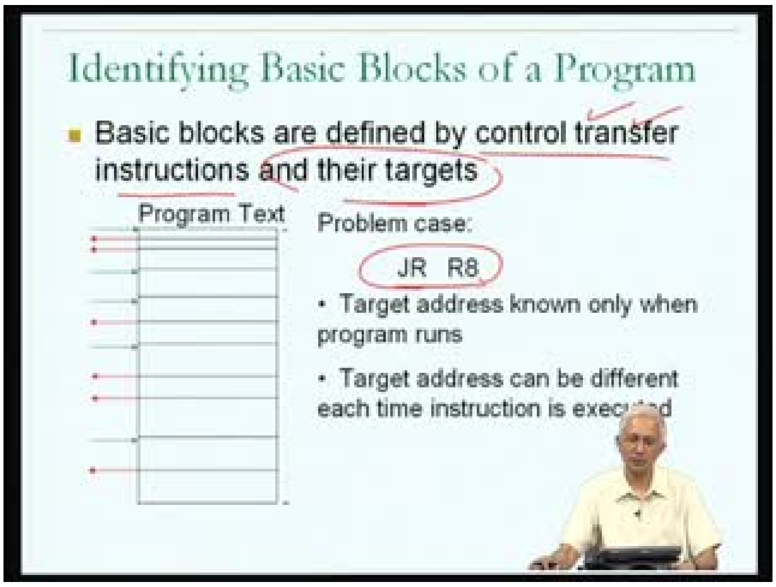
Pixie: How it works

1. Identification of basic blocks
   - Q: How can basic blocks be identified?
   - Pixie uses heuristics where necessary
2. Instrumentation

Now, let me explain how pixie works in terms of the instrumentation that it does now very clearly you will remember that under the pixie frame work, pixie takes in pixie is the name of the program that takes in an executable file. And as instrumentation into that program in order to count the number of times that each basic block of the program is executed. So, very clearly one thing that pixie has to do is identify the basic blocks of a program, and the other thing that it has to do is it has to add instrumentation into the program in order to count the number of times that each basic block is executed.

So, the first question in understanding how pixie works, is how can basic blocks be identified and we are going to look into this little bit, but it turns out that pixie used certain heuristics where necessary we are mean by heuristic is some kind of a guideline not based on any exact scientific principle where just a guideline.

(Refer Slide Time: 37:56)



Identifying Basic Blocks of a Program

- Basic blocks are defined by control transfer instructions and their targets

Program Text

Problem case:

JR R8

- Target address known only when program runs
- Target address can be different each time instruction is executed

So, let us try to understand how to identify the basic blocks of a program. Now, we understood that the basic blocks of a program are defined by the control transfer instructions and the targets of control transfer instructions. Therefore, we could infer that if I could identify all the control transfer instructions in a program, and I could also identify all the targets target addresses of the control transfer instructions in a program. Then I would know the basic blocks of the program and this gives me the functional mechanism to identify the basic blocks of a program.

So, let us suppose that I have a program text. So, these are the instructions of the program from the first instruction going till the last instruction, and I know that I can identify the control transfer instructions. Because if I examine each instruction of the program, and I know the instruction format then, I can write a program which could decode the instruction, and determine if it is a control transfer instruction or not.

So, it should be possible to identify the control transfer instructions. So, if I identify the control transfer instructions I could mark them. So, here I have in red, I have arrows which are pointing away from each of the control transfer instructions in the program. So, we have done the first I have identify the control transfer instructions next I need to identify all the targets of control transfer instructions in other words the different places to which a control transfer instruction might transfer control. And once again if I have each of the control transfer instructions I could look at the control transfer instruction to identify its target and I could mark those in the program.

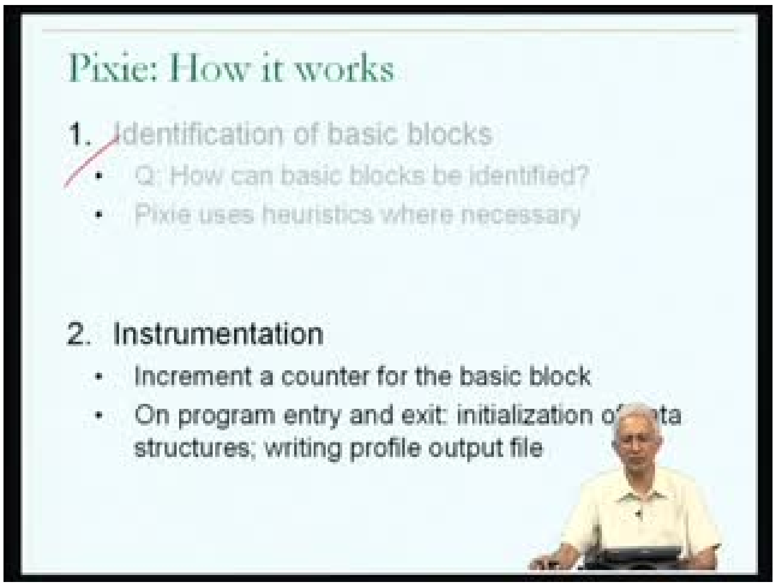
So, in the green arrows that I have put show you the targets of the different control transfer instructions. And therefore, to identify the basic blocks all that I have to do is I could say the region in the program between any two of these arrows gives a basic block. And that is what I could do I could mark the regions between these targets of control transfer instructions, and the control transfer instructions themselves and say those are the basic blocks of the program that is sounds easy enough.

There is of course, one problem and that is it may not be possible for me to actually identify all the targets of the control transfer instructions. Let us just look at a simple example from our MIPS 1 experience here is the control transfer instruction of the MIPS 1 jump register R8. So, this transfers control to the instruction whose address is inside register R8.

Now, as this program executes it is possible that different values get into register R8. And unless I know all the different values that could get into register R8 I could not possibly know the different targets of this control transfer instruction. And therefore, there are certain problem cases even for the simple MIPS 1 instruction set, which make it practically impossible for us to actually identify all of the targets of control transfer instructions.

Identifying all of the control transfer instructions is not a problem because we know the instruction set with this is why pixie had to use certain heuristics, but you could sort of understand that for the different possible c constructs, which may result in control transfer instructions such as while loop repeat loops switch statements etcetera. It might be possible for heuristics to be identified in the event that this kind of an instruction has to be used in the compilation of that particular c construct.

(Refer Slide Time: 41:07)



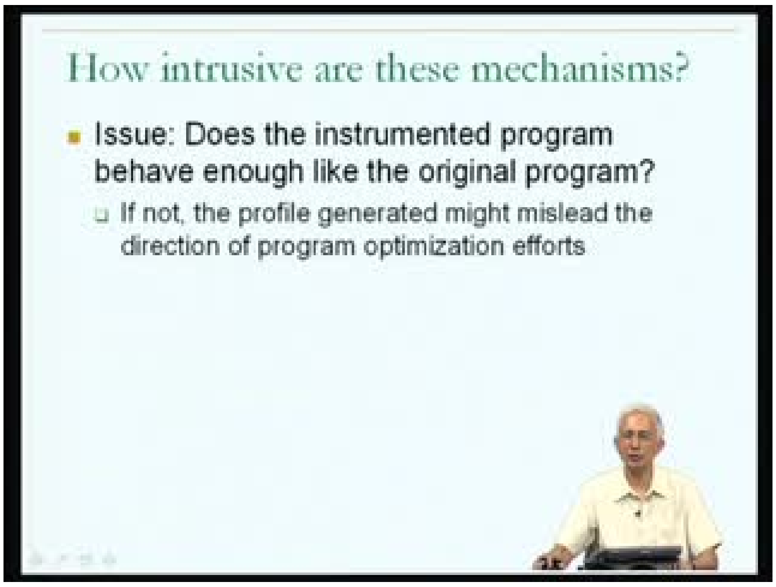
Pixie: How it works

1. Identification of basic blocks
   - Q: How can basic blocks be identified?
   - Pixie uses heuristics where necessary
2. Instrumentation
   - Increment a counter for the basic block
   - On program entry and exit: initialization of data structures; writing profile output file

That's why c heuristics may have been necessary in the case of pixie so this was the problem with a jump register instruction. So, we understand how the identification of the basic blocks could be done the next question is what instrumentation would pixie do in other words what would it have to do to generate the instrumented version of a dot out.

And basically remember that all the pixie does is it gives us basic block counts the number of times that each basic block count is executed. Therefore, all that has to be done is increment a counter associated with each basic block. So, at the beginning of each basic block include instructions to increment a counter for that basic block initialize all the counters to 0 prior to that, and at the end of the program execution cause the all the counters to be printed into the output file.

(Refer Slide Time: 41:51)


How intrusive are these mechanisms?

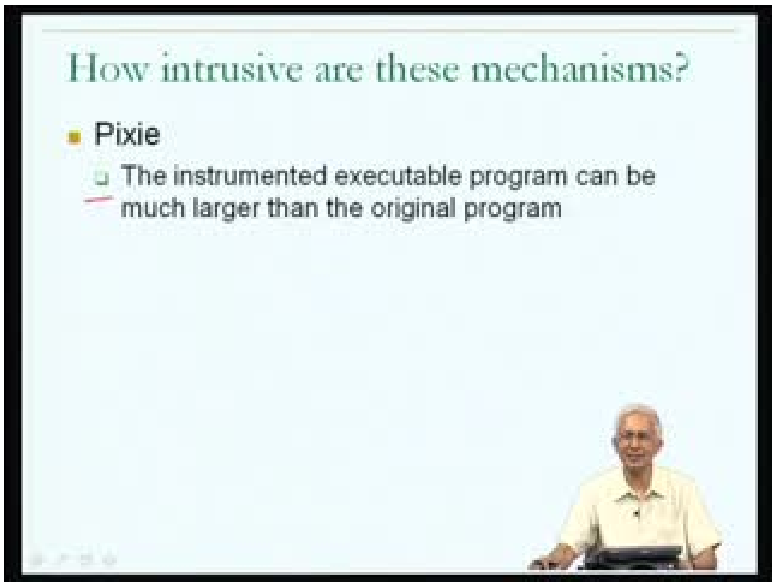
- Issue: Does the instrumented program behave enough like the original program?
  - If not, the profile generated might mislead the direction of program optimization efforts

So, one program entry and exit initialize the data structures and write profile output file. So, the instrumentation seems to be relatively simple. Now this point we need to ask one question and that relates to what I will call the intrusiveness of such profiling mechanisms. We have seen two profiling mechanisms, one was the function level profiling mechanism prof; the other was the basic block level profiling mechanism which I referred to by the name pixie.

And I arise a question of how intrusive are these and the underlying issue, which we were trying to address is does the instrumented program behave enough like the original program. Now just remember that in both cases both the case of prof, and the case of basic block level profiling using pixie, we did not use the a dot out file corresponding to the program that we were interested in.

In both cases, we ran an instrumented version of our program. And the instrumented version was not exactly the same as our program by definition because it contained instructions which were doing the instrumentation or the measurement of the breakup of time or the measurement of number of times that the basic block level was executed. Therefore, by definition the instrument the program that we ran in the profiling run was different from the program that we are interested in, and that therefore, if they are sufficiently different then they may not behave alike and that one has to worry a little about whether the difference between the two programs. The so, substantial that the intrusiveness has become a problem, hence the issue is one of relevance we have to satisfy ourselves that neither of these mechanisms is too intrusive. Because if it is too intrusive then the profile generated might mislead the direction of our program optimization efforts.

(Refer Slide Time: 43:33)

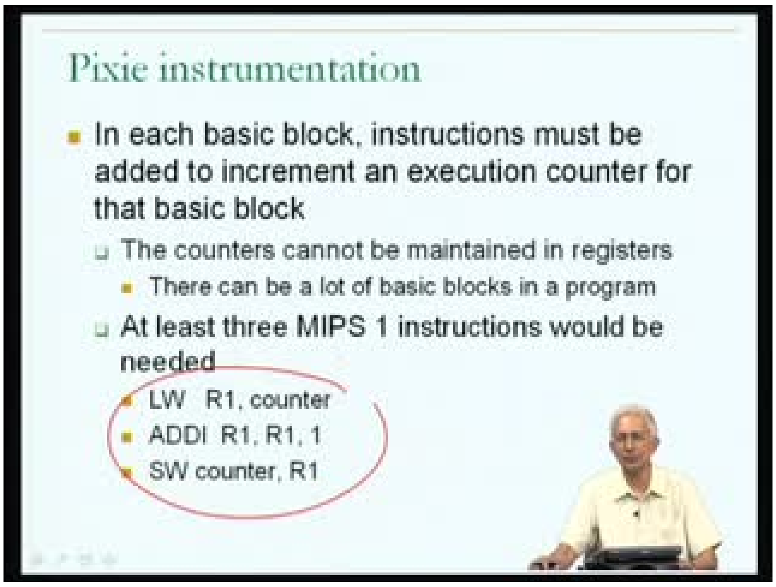


How intrusive are these mechanisms?

- Pixie
  - The instrumented executable program can be much larger than the original program

And therefore, would be counterproductive for us to use the profiling mechanism. So, we need to adhesive the question of how intrusive are these mechanisms. Let me start by talking about pixie, now one thing we should note about pixie is that the instrumented executable program could be much larger than the original program. Is this a problem now? You should note that if the instrumented program is much larger than the original program, then it could suffer many more page faults it could suffer many more cache misses and therefore, could end up having different timing properties than the original version of the program and therefore, this could be a concern, but let us first of all satisfy ourselves that this is the problem with pixie.

(Refer Slide Time: 44:13)



Pixie instrumentation

- In each basic block, instructions must be added to increment an execution counter for that basic block
  - The counters cannot be maintained in registers
    - There can be a lot of basic blocks in a program
  - At least three MIPS 1 instructions would be needed
    - LW R1, counter
    - ADDI R1, R1, 1
    - SW counter, R1

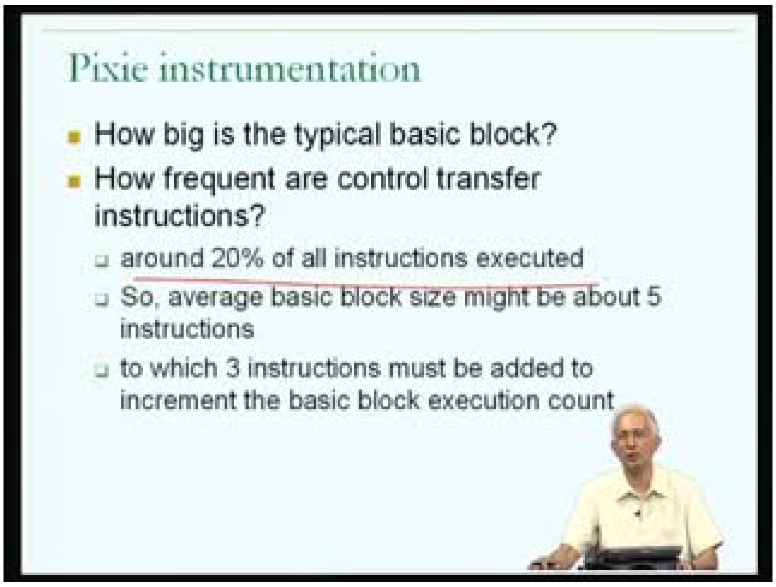
Is that the case that the instrumented executable problem could be much larger than the original program. Now thinking back to what pixie did, we understand that it instruments each basic block by adding instructions to increment an execution counter for that basic block that is all that it did as far as the each basic block was concerned. So, we have to understand how many instructions it would have add to add into each basic block in order to do this activity how many instructions would be required a counter for that basic block.

Now, the first thing to note is that these counters cannot be maintained in registers because the number of basic blocks in a program could be fairly large. And therefore, the counters must be maintained as a similar to the way profile does it using memory.

And therefore, in order to increment a counter for a basic block they would have to be at least three MIPS 1 instructions. one to load the counter into a register; one to increment in that register; and another to store the value of the incremented counter into the memory location. Therefore, we were suspect that for something like the MIPS instruction set the instrumentation would amount to 3 instructions for each basic block.

Now, the next question is that a lot now that is going to depend on the size of the basic block. If the size of the each basic block is 50 instructions, then adding 3 instructions may not be that much of a problem. If the size of each basic block is 3 instructions then adding three more instructions to it is going to double the size of the basic block and that would a bit of a problem, or that could be a bit of problem. Therefore, to understand if instrumentation which requires 3 instructions to be added to each basic block is too much, we have to know a little more about how big each basic block is.

(Refer Slide Time: 45:52)



Pixie instrumentation

- How big is the typical basic block?
- How frequent are control transfer instructions?
  - around 20% of all instructions executed
  - So, average basic block size might be about 5 instructions
  - to which 3 instructions must be added to increment the basic block execution count

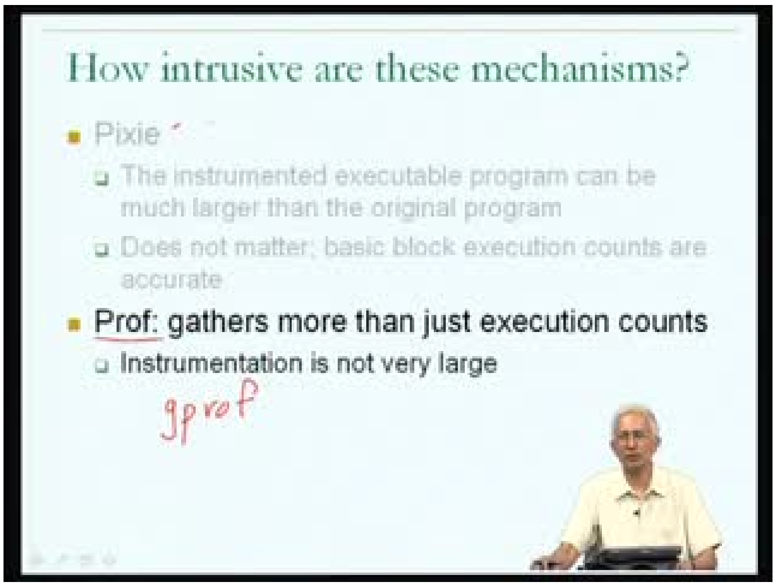
So, what is the size of a typical basic block and this will of course, vary from program to program, but we could get some rough idea about how big each basic block is if we had some idea about how frequently control transfer instructions are executed.

For example, if we knew that let say 20 percent of all the instructions executed by a typical program are control transfer instructions. And this kind of an information might be available in different sources of information that is suppose this is the case that how could be from this get some idea about the average size of a basic block if 20 percent of all the instructions executed are control transfer instructions; that means, a1 in 5 instructions which are executed are control transfer instructions, from which we could infer that the average size of a basic block is something about 5 instructions 56 instructions.

And if this is the case, in other words if around 20 percent of all instructions executed are control transfer instructions, then we could actually infer that adding 3 instructions into a basic block is going to increase the size of the average basic block by about 50 percent and that might be a concern, where increasing the average size of a basic block from about 5 to about 8 and therefore, the size of the program is going to go up by again by about 50 percent.

Now, the question of how do we assess whether this intrusiveness is bad, and makes pixie a mechanism that we should be suspicious about, but bear in mind that all pixie is doing is giving us basic block execution counts and the accuracy of the count is not going to be effected by the size of the program, even if the program becomes hundreds of times of larger and suffers huge amount of cache misses and page faults the accuracy of the count is not going to be effective.

(Refer Slide Time: 47:34)



How intrusive are these mechanisms?

- Pixie
  - The instrumented executable program can be much larger than the original program
  - Does not matter; basic block execution counts are accurate
- Prof: gathers more than just execution counts
  - Instrumentation is not very large

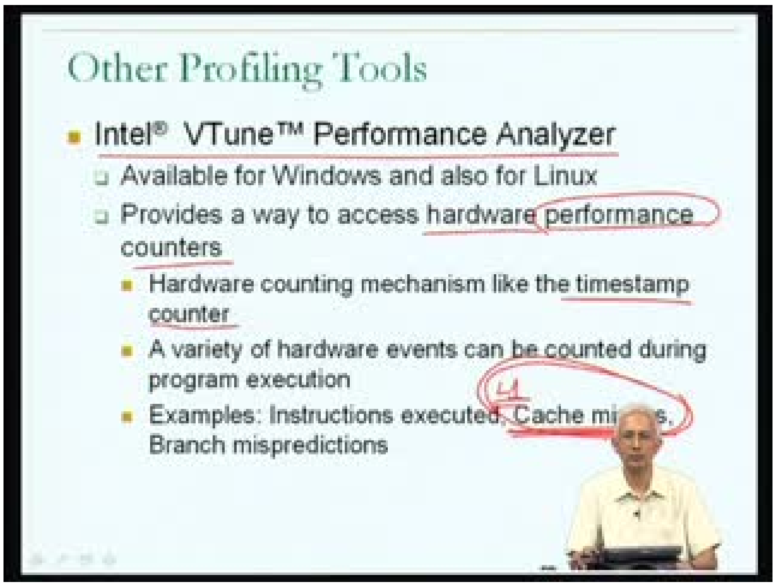
gprof

And therefore, we do not have to worry about pixie as far as the accuracy of the counts is concerned it does not matter because the basic block execution counts are accurate. So, pixie instrumentation will increase the size of the program possibly substantially, but we do not have to worry about it, because it is does not affect the accuracy of the counts. Now what about prof, prof is a little different in the prof is trying to give us accurate not just accurate execution counts, but also breakup of execution time.

Now, in the case of prof if you think back to what we discussed the size of the instrumentation is not very large what was the instrumentation in the case of prof what it did was at the beginning of each function it added instructions to increment a counter for the function. So, 3 instructions MIPS 1 instructions would be added to the each function, and we understand that the average size of a function is going to be substantially more than the average size of a basic block. And therefore, adding 3 instructions into each function should not be viewed as being large, other than this all that the instrumentation for prof did was to include the system call profile, and then at the exit point of the programs to cause the counters which had been accumulated to be return to the output file mon dot out.

So, in the case of profit we do not have any reason for concerned, because the instrumentation is not very large in size and it is not going to change the properties of the program much for that reason. So, with this we have seen these two mechanisms, which are available on mechanisms of these kinds are available on most of the computer systems that you will encounter prof by the very same name variance on prof such as something called g prof, which will give you another level of information about the relationship between the functions that are called, and in addition to this the pixie type basic block level mechanisms some of which are available directly from instrumentation done by the compiler.

(Refer Slide Time: 49:33)



Other Profiling Tools

- Intel® VTune™ Performance Analyzer
  - Available for Windows and also for Linux
  - Provides a way to access hardware performance counters
    - Hardware counting mechanism like the timestamp counter
    - A variety of hardware events can be counted during program execution
    - Examples: Instructions executed, Cache misses, Branch mispredictions

Now, are there any other mechanisms that we should be aware about that are widely available on computer systems that might be useful to us in getting ideas about the important parts of our programs, to other important profiling tools I would like to mention just one there are several, but I like to mention one and that is the Intel VTune performance analyzer. So, this is a product available from Intel, and this is available both on windows as well as for Linux, for Linux systems and basically this is special in that unlike the previous two mechanisms, that I have talked about in other words prof and basic block level profiling using something like pixie.

VTune provides a way to access something called hardware performance counters. Now hardware performance counter as a name suggest are mechanisms that are available in the hardware for us, to get some count information about different performance aspects of what is happening in the hardware. When our program executes and from this we should in understand that in addition to hardware counting mechanisms like the timestamp counter, which we saw little while back when we are talking about timing modern processors actually contain a variety of other hardware events, they can be counted during program execution.

And examples of the kinds of hardware events that modern processors allow us to keep track of a things like the number of instructions, that I have been executed the number of L1 cache misses that have occurred, or even other kinds of information which we not entirely aware of. But you can well imagine that if you are able to get information about something like this the number of L1 cache misses that occurred. When your program was executing then this would be of great value to us in understanding the cache behavior of our program even better, which is why a mechanism like Intel VTune performance analyzer which were provides access to information not only about cycle counts, but also to other kinds of events such as cache misses would have additional value to a programmer in understanding what are the important parts of the program, and in analyzing what kinds of activities should be attempted to improve the nature what is happening in that part of the program.

Now, with this we have had a quick look at the some of the aspects of the different mechanisms are available to get timing and performance, understanding of what is happening when our programs execute. We understood that there are all the UNIX and Linux systems, we are likely to use function level profiling using something called prof is available and that are many systems, that we are going to have access to basic block level profiling is also available to get information back at a much lower level in terms of the frequency with which or the number times that basic blocks which are 5 or 6 instructions in size on average were executed when our program ran.

And in addition to this that there are commercially available tools such as Intel VTune performance analyzer, through which we could get information of a much more sophisticated nature possibly relating to the number of times that cache misses occurred during a particular function execution of relevance to us.

So, with this we understand that variety of tools are available in connecting with profiling, and timing and that there are exactly what are required for us to get better understanding about the way that our program is interacting with a hardware. And the operating system and how therefore, point us to directions that we could take to improve the qualities of these programs. Thank you.