

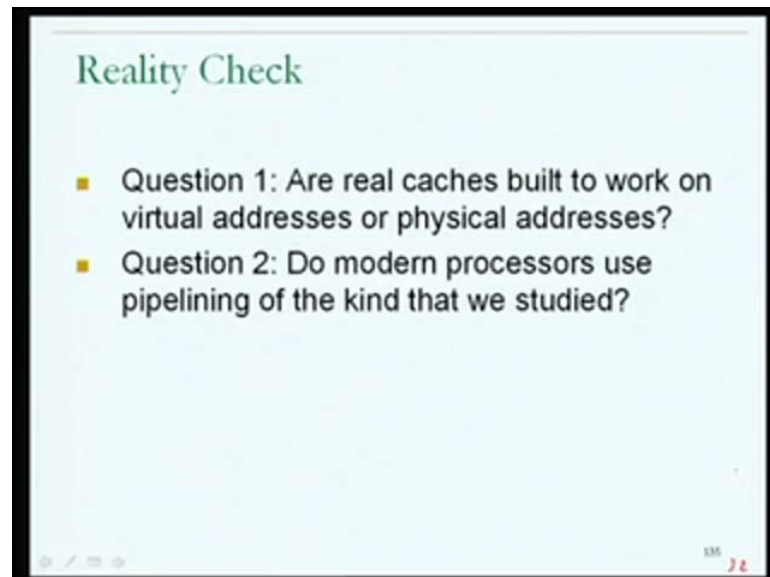
High Performance Computing
Prof. Matthew Jacob
Department of Computer Science & Automation
Indian Institute of Science, Bangalore

Module No. # 07

Lecture No. # 32

Welcome to lecture 32 of the course on high performance computing. You will remember that in the previous lecture we had looked at many examples of small pieces of C program code and try to understand how they would perform for different kinds of cache organizations. And, we got some understanding of how one could improve the quality of a loop in order to benefit from the spatial or temporal locality that of reference through which the performance with the cache would be improved.

(Refer Slide Time: 00:43)

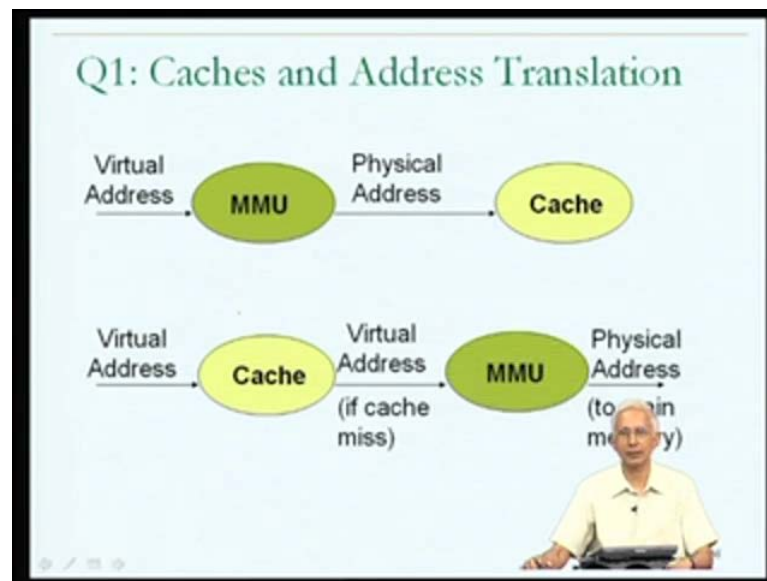


Before moving into the next topic, I wanted to just quickly address two issues relating to caches; one relating to caches and one relating to pipelining which give us slightly better idea about what happens in the processors which are currently quite popular. The first question relates to caches in the sense of we talked about the C P U sending an address to the cache for look up and subsequently if it is a hit, the data would come from the cache.

But the question has to whether it was virtual or physical addresses that are used by the cache is of relevance.

So we are looking into the relationship between the addresses that the cache uses, the cache hardware uses and the address translation; that we now understand the operating system manages but is done by a piece of hardware. So we understood that there are two possibilities as to how a cache could be built.

(Refer Slide Time: 01:32)

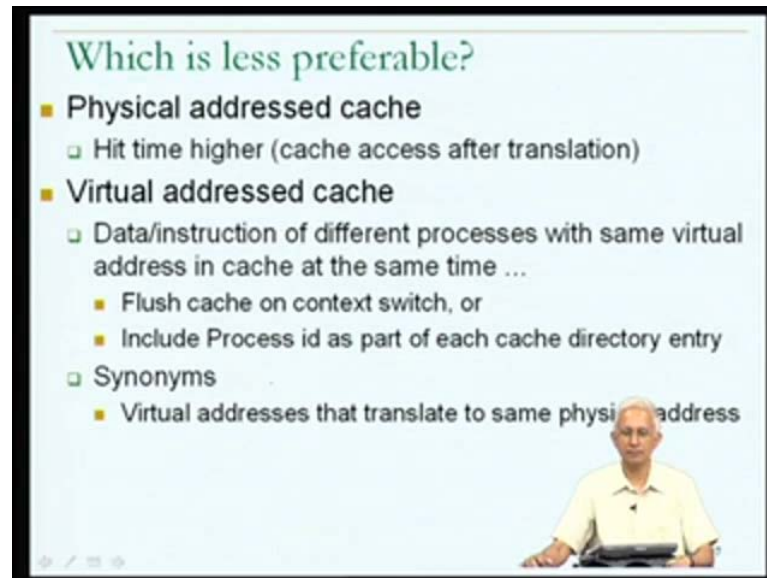


The first is under the assumption that the cache is actually based on physical addresses; in other words after the processor C P U generates an address, it is translated by the memory management unit into a physical address which is what goes to the cache. The second possibility is that the address which is generated by the C P U is directly used by the cache hardware; in other words the virtual address is what the cache hardware uses for its look up tag checking and so on.

These are two different kinds of caches, you will note that in the second option there is the possibility of a cache miss and in the event of a cache miss the data will have to be fetched from main memory for which a physical address will be necessary. Therefore, in the event of a cache miss in the second option, there would be a need for the address translation to happen in order to generate the physical address for fetching the cache, the missing block from main memory into the cache.

So, these are the two kinds of caches we might expect to find. And, I refer to the upper kind of a cache as a physical address cache because it is organized based on physical addresses. The cache hardware is organized to take in physical addresses. And, the second, the lower example lower possibility is what I refer to as a virtual address cache. And, we look at some of the negative aspects of both of these in the previous lecture.

(Refer Slide Time: 02:53)

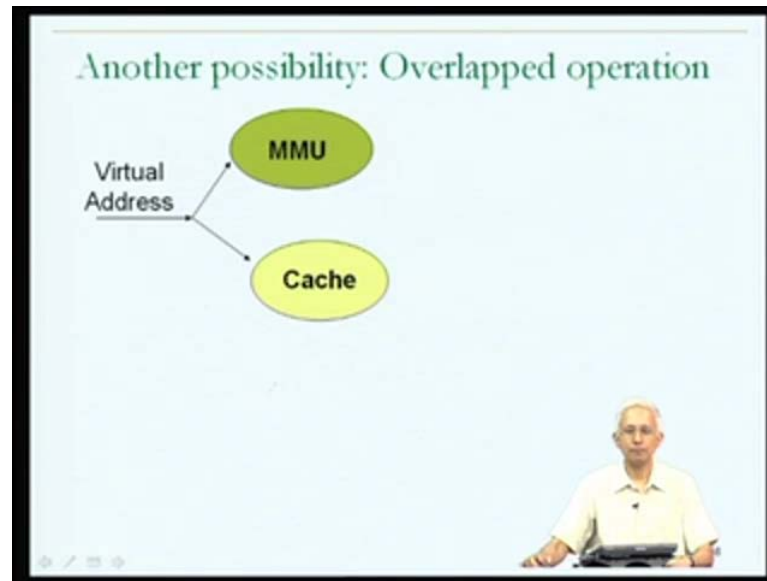


Which is less preferable?

- **Physical addressed cache**
 - Hit time higher (cache access after translation)
- **Virtual addressed cache**
 - Data/instruction of different processes with same virtual address in cache at the same time ...
 - Flush cache on context switch, or
 - Include Process id as part of each cache directory entry
 - Synonyms
 - Virtual addresses that translate to same physical address

We understood that there are some benefits of the virtual address cache, in that the time of a cache hit would be less than that for a physical address cache, partly because translation does not have to be done but also for other reasons. But in both cases there are some negatives to the use of the cache organization, the physical and the virtual address cache; which nears open the possibility that there is another option which might in fact be a good idea. And, the what the third option would try to do is to try to have some of the positive aspects of the physical address cache and some other positive aspects of the virtual address cache. And, mechanism I am going to describe something which will just try to use overlapping; in other words it will try to do in parallel some of the steps associated with address translation and cache look up. Now, remember that in both the scenarios that we had setup either the virtual or the physical address cache the processor generates a virtual address.

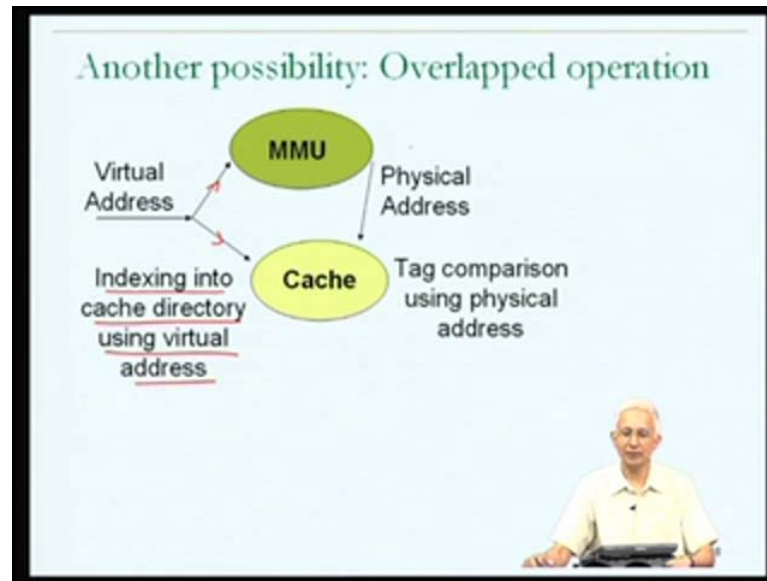
(Refer Slide Time: 03:54)



Now, in the first option that we look at the virtual address went directly to the MMU for translation and the second option the virtual address went directly to the virtually addressed cache. But we could have a third option in which the virtual address goes to both, both to the MMU for translation and to the cache for look up. Now, in the event there is a cache miss, we will still need to have the translation completed and that is the reason that the virtual address is sent both to the MMU and to the cache, but these two are going to be happening in parallel. So, while the translation is being done by the MMU, the early parts of the cache look up can happen within the cache.

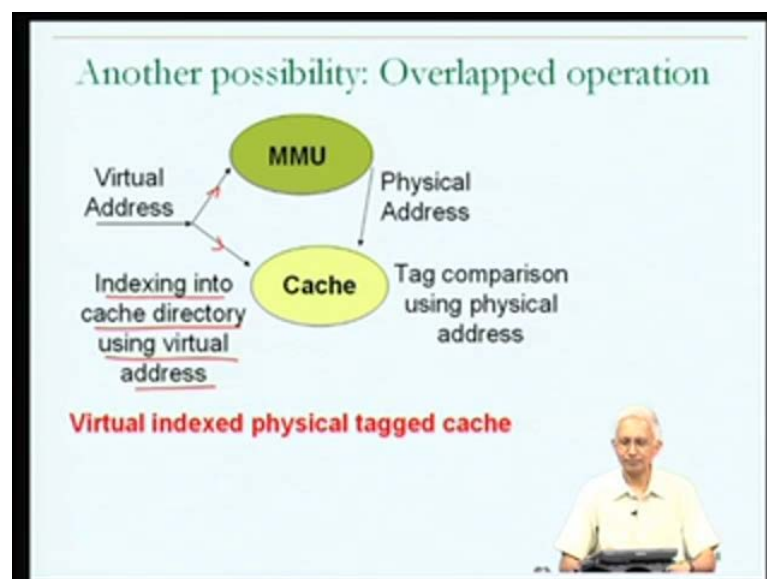
Subsequently by the time the address translation is completed and the physical address is available, the physical address can be used for the subsequent activities within the cache. For example, if by the time the translation has been done, the cache look up in other words indexing as already been done, then with the physical address that has been made available by the MMU due to translation being completed, the tag checking could be done using the physical tag.

(Refer Slide Time: 05:00)



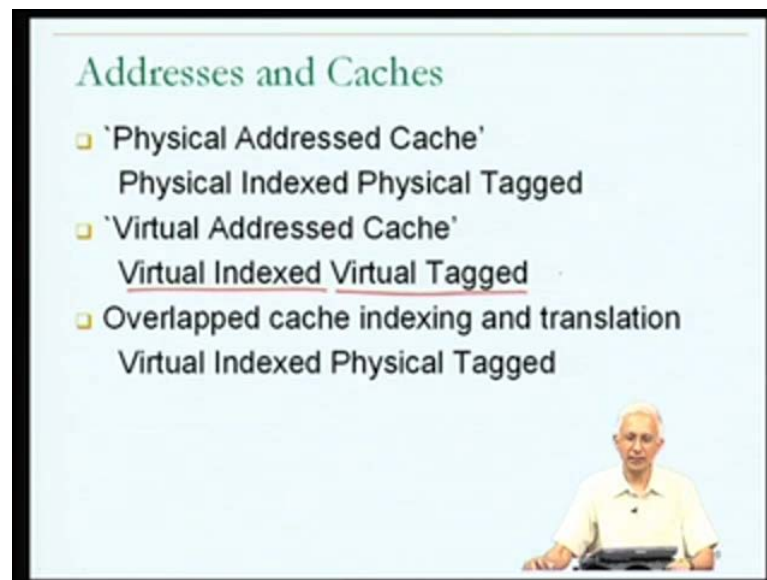
And, that is the idea which I am showing over here the idea that the virtual address goes to both the MMU and to the cache. While the virtual address is being used do indexing into the cache directory, it is also being used in the MMU for translation. So that not too much time later the physical address is available and the physical address can be used for the tag comparison. This will allow us to have something which is neither option 1 that we saw in the previous slide nor option 2 in the previous slide, but something new. And, this in fact has some benefits so over either option 1 or option 2.

(Refer Slide Time: 05:37)



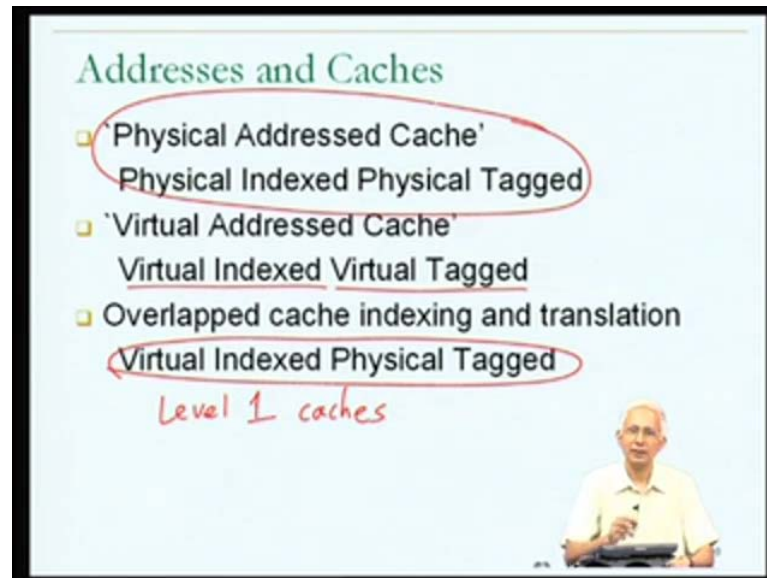
And, I will refer to this as I cannot prefer to this as either a physical address cache or a virtual address cache. Since, it is both neither and both; it uses a virtual parts of the virtual address for indexing and it uses parts of the physical address for a tag comparison. Therefore, the only good name for this kind of a cache would be to elaborate and say that it is a cache which uses virtual indexing and physical tagging. In other words it uses the virtual address bits to index into the cache directory, but it uses the physical tag due to the tag comparison. So, this kind of a cache would be known as a virtual index physically tags cache, suggesting that the other alternatives that we had looked at could have similar names.

(Refer Slide Time: 06:19)



And, in fact instead of refereeing to a cache as a physical addressed cache, when am talking it as a physical index physical tag cache. What we call a virtual address cache up to now, in our second option might more accurately be described as a virtual index virtual tag cache, in the sense that the virtual address cache that we looked at used virtual address bits to index into the cache directory; it also used virtual address bits to do the tag comparison. Whereas, the third option that we are just seen is what we call a virtual index physical tag cache. Now, as it happens the third option is what is used to be in the virtual index physical tag cache, is what is used in the vast majority of processes today for the level 1 caches. Remember that we understand that in processors today it is we expect to find both level 1 caches which are relatively small but extremely fast.

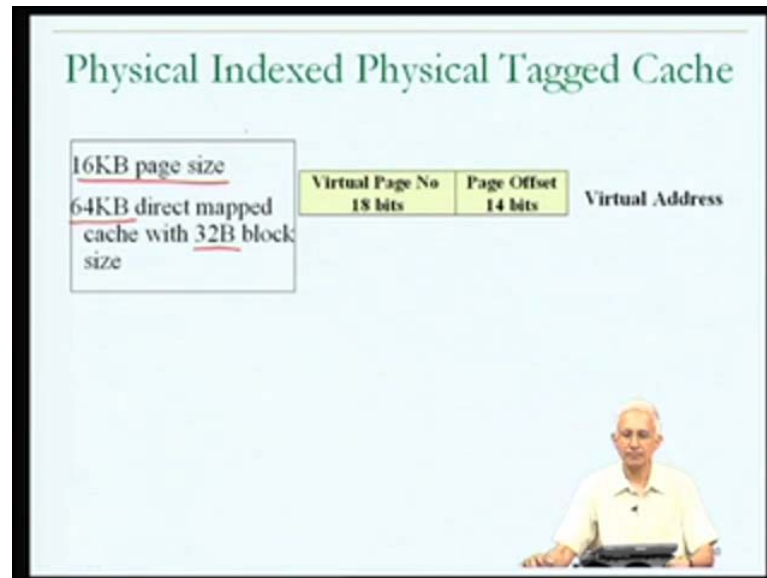
(Refer Slide Time: 07:12)



But backed up by level 2 and may be level 3 and level 4 caches which will be larger but slower and this is done to offset the speed disparity, the growing speed disparity between processors and memories. So, the indication that I am giving is that in today's processors so level 1 caches actually use virtual indexing and physical tagging. The part of the reason for this is likely to be because we saw that the physical address caches suffer from large hit time; it takes a long time to get a hit in the cache because the translation has to happen before the cache is refer to. On the other hand, the virtual address caches have a much smaller hit time and the virtual index physical tag cache will have the same benefit. It will not have many of the negatives associated with either of the first two options.

So, it is preferable option and it widely used in the level 1 caches of today. Just to give us a better understanding of how these three differ. I will just go through this diagrammatic mechanism for showing it what happens to an address in each of the three cases.

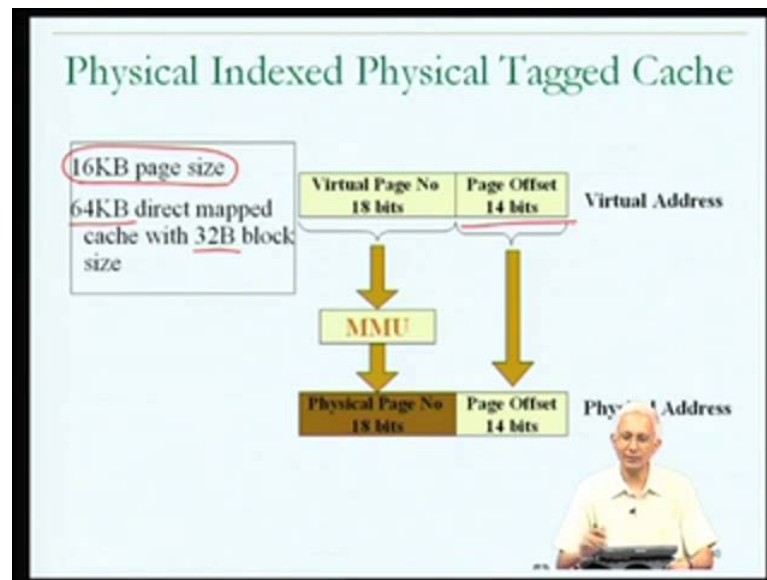
(Refer Slide Time: 08:20)



And, the numbers which I am going to be using for the different bit fields, the size of the different bit fields we will assume a 64 kilobyte direct map cache which has 32 byte blocks. In addition, since we are also going to be talking about address translation, remember that address translation happens on the, involves mapping of a virtual page number into a physical page number. So, suddenly it also becomes important to know something about the page size. So, this will also enter the calculations in our, into the, be a factor in the diagrams that we are going to look at.

So, we will start by looking at a physical index physical tag cache and when we think of a virtual address, we from the perspective of address translation recall that a 32 bit virtual address would have least significant bits that we call the page offset bits and most significant bits which we call the virtual page number from our discussion of virtual memory. How many bits are used for the page offset? That would be determined by the page size. So, if the page size is 16 kilobytes they would be 14 bits of page offset, 14 is log base 2 of 16 k. The remaining 18 bits are the virtual page number. Now, if you are talking about a physical index physical tag cache, then this address will have to be translated by the MMU before it goes to the cache.

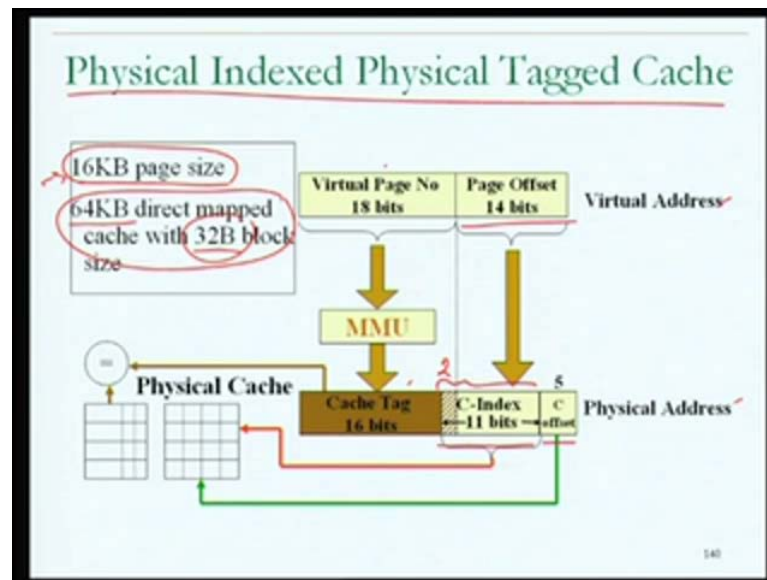
(Refer Slide Time: 09:32)



So, what is the MMU do? The MMU actually translates by replacing the virtual page number by a physical page number and the page offset bits, the least significant 14 bits are not affected by the translation. Therefore, if I show you what the MMU does, it would amount something like this; it translates the virtual address into a corresponding physical address. The virtual and physical addresses are identical in the least significant 14 bits for this particular example. And, they differ only in the most significant 18 bits because what that MMU did was it replaced the virtual page number by the corresponding physical page number, using the information inside the relevant page table entry; possibly available in a translation look a side buffer which is a part of the processor.

Where so, this is the address, the physical address is what is going to be used to access the cache, in the physical index physical tag cache and it is this address which we have to look at in the light of the cache perspective on addresses.

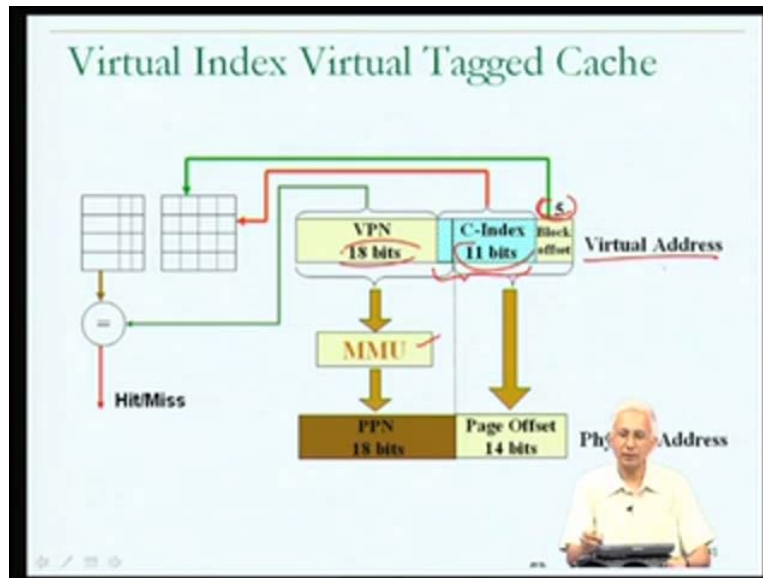
(Refer Slide Time: 10:30)



So, this is now going to be viewed as a cache block offset which is going to be 5 bits in size because the block size is 32 bytes. It is a direct map cache of 64 kilobytes size with 32 byte blocks from which we can calculate that the number of index bits required is going to be 11 and the remaining bits which in this case will be 16 you will be the tag bits. So, the 11 index bits are used to index into the cache or the cache directory and then the tag comparison is done. The valid bit is also refer to and if all of these indicate that this is the cache hit, then the offset bits are use to select the correct bytes out of the cache ram to satisfy the processors request.

So, you will notice that for this particular example the index bits are coming partly from the page offset; just look at where the eleven bits are coming from. The bulk of the them are coming 9 of them are coming from the page offset bits which are common to the physical address and the virtual address and only 2 of the bits are coming from what is now the physical page number side of the address bits.

(Refer Slide Time: 11:35)



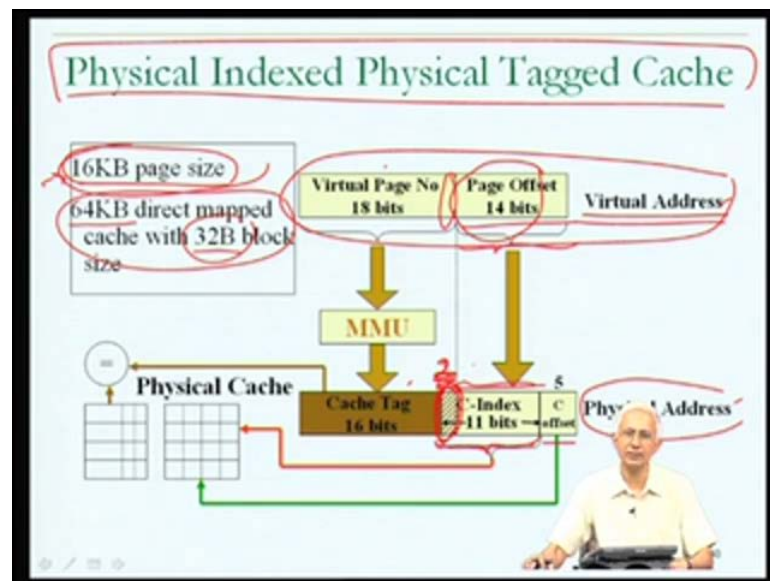
Now, if we now move to the second possibility that we had, the virtual index virtual tag cache, which is what we were calling the virtual address cache until now. So, in this case the address translation may not happen; address translation is not required if there is a cache hit. So, the virtual address itself is viewed in the light of a 5 bits of offset, 11 bits of index and 18 bits of tag. The index bits are used to index into the cache directory and then the virtual page number provides the tag bits which are use for tag comparison and if the indications are that it is a hit, then the correct bits are picked out using the offset.

Now, if it is the cache miss then the MMU does a translation and we get the physical address which goes to memory. So, in this case once again you will notice that all of the 11 bits which correspond to the index, 9 of them are coming from the page offset bits and the remaining 2 are coming from the least significant bits of the virtual page number. Now, you may be wondering why am I pointing out where the bits of the index are coming from. This relates to something that we were doing in the early parts of the previous lecture. You recall that when we were talking about how to figure out the number of hits, the number of misses for a loop that is of interest to us from a program, that we may be running on a computer. We made some we used information about the addresses of the arrays array a or array b etcetera.

In order to calculate what the index bits are in order to get an idea whether they were conflicts hits or misses etcetera. And, at that point we were assuming that the cache was

indexed using virtual addresses; you are assuming that the addresses which the compiler generated in other words the virtual addresses are what we being used for access in indexing into the cache and for the tag comparison. And, that is a perfectly legitimate assumption as far as the virtual index virtual tag cache is concerned, because it is virtual addresses that are use for this purpose in such a cache; but is as we now see not entirely correct as far as the physical index physical tag cache is concerned.

(Refer Slide Time: 13:30)



I have gone back to the previous slide, so you will note that in our calculations in our estimates for the various kinds of small examples that we looked at, we did the calculations assuming that they were virtual addresses. But in reality the cache was indexed into and the tag comparison was done using physical addresses. If we are looking at example of the physical index physical tag cache and what difference would have made in our estimates of hits and misses, we notice that the index the primarily the calculations that we were doing in looking at the reference sequence that we had and so on, was to try to find out what the index bits were for a given memory reference.

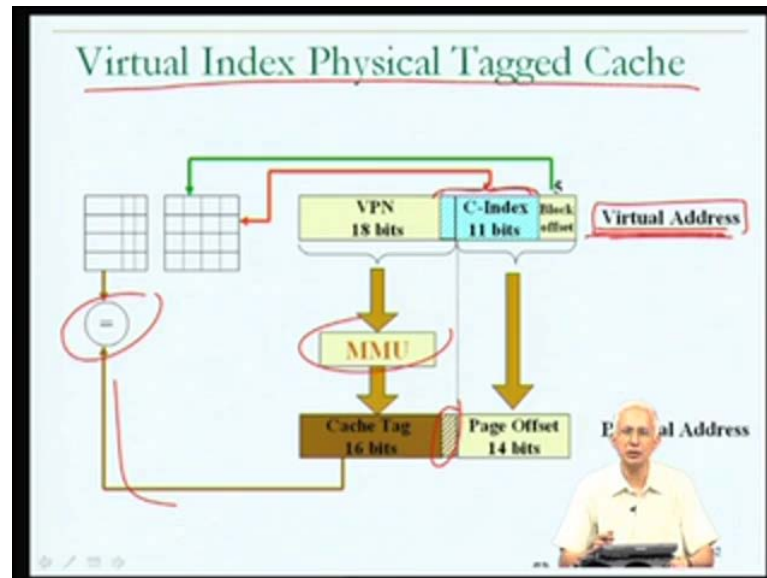
We now see that for the example that we have for the sizes that we are looking at here of the 11 index bits 9 of them are coming from the page offset region of the address and 2 of them are coming from the virtual page number or physical page number size of the address, which means that when we made the assumption that we were using virtual addresses; they would have been some to bits in those 2 positions. But in reality when

the cache is looked up, they will be some other 2 bits depending on how the address translation maps the virtual page number into the physical page number. And, therefore, a calculation of those index values would have been wrong in the case of the physical index physical tag cache, because we were working with virtual addresses; we had no means of knowing what the physical addresses work.

The physical address of a particular page will change as the page moves in and out of memory depending on the occurrence of page faults, we have no means of knowing with the physical addresses are. However, once again if you think about what we were doing in our calculations, we were trying to cause offsets to happen to prevent index bits from being exactly the same. And, these were in situations where we were concerned about neighboring or contiguous regions in memory, such as the example we had of the array a and the array b I am sorry the vector a and the vector b which had assumed base addresses of x a 0 0 0 and e 0 0 0 0. And, under the assumption that these are actually fitting on the same page, you will notice that there is no problem.

So, if the data that we are talking about but actually fit on the single page, then this problem does not arise at all. Because all the difference in the different addresses will occur within the page offset bits and therefore, there is no problem as far as we are concerned from that perspective. However, if the **problem**, if the data is larger and actually causes the difference in the index bits to happen between these 2 bits, then the calculations that we were doing would be not precisely correct. And, therefore, we have to understand what we were doing is just estimating the behavior as far as the hits and misses were concerned in the, in case of this being a word physical index physical tag cache.

(Refer Slide Time: 16:14)



Now, moving to the third option that I just introduced in this lecture, which I referred to as the virtual index physical tag cache; you will recall that the virtual address is what is used to index into the tag directory I am sorry to index into the cache directory. So, it is the 11 bits from the virtual address that are used to index. Subsequently, the address translation happens and we get a physical tag which is what is used for the tag comparison; the physical tag is use for the tag comparison. And, this is what will happen in the case of the virtual index to physical tag cache.

So, this is what we actually understand is going to be present in bulk of the l1 caches in processors today. And, this two will have the problem that we referred to, in terms of the tag bits will be a little bit different if we did our calculations for the behavior of our programs based on virtual addresses. However, since the indexing is going to be using virtual addresses or calculation of index conflicts is going to be accurate. And, therefore, there was no need for us to worry too much about the technique that we used in the case of analysis of the simple loops, if we were dealing with virtual index physical tag caches. Since, our calculation of which different pieces of data like the vector a, the vector b whatever; would actually have conflicts as far as the index bits is concerned, what have been accurate in the light of I have using virtual addresses, if it was a virtual index physical tag cache.

So, hence this additional information would be important to us if for example, we get more details about the nature of the L1 caches in the processors that we are dealing with. And, the techniques we were using a perfectly legitimate for a virtual index physically tag cache or virtually index virtually tag cache, but were only be estimate as far as a physical index physical tag cache is concerned. Now, moving right along, also wanted to make a few comments about the second question; which will give you; give a rough idea about a few other developments that have happened in the area of pipelining. So, we talked about pipelining in the light of the 5 stage example there was the pipeline where there was instruction fetch instruction decode, execute memory operation and write-back. And, the question is modern processors really built using pipelines of that kind.

(Refer Slide Time: 18:30)

The slide is titled "Q2: High Performance Pipelined Processors". At the top, five stages of a pipeline are shown in circles: IF, ID, EX, MEM, and WB. Each circle has a red arrow pointing to the right, indicating the flow of instructions. Below the title, there are three bullet points:

- Pipelining
 - Overlaps execution of consecutive instructions
 - Performance of processor improves
- Current processors use more aggressive techniques for more performance *ILP*
- Some exploit **Instruction Level Parallelism** - often, many consecutive instructions are independent of each other and can be executed in parallel (at the same time)

In the bottom right corner of the slide, there is a small video inset showing a man in a light-colored shirt, presumably the presenter, sitting at a desk.

Now, it turns out that the kind of pipelining which we saw was getting performance benefit by overlapping the execution of many consecutive instructions; that is why we had 5 stage pipeline you will recall our notation from many lectures back. The 5 stages one for fetching instructions, one for decoding instructions and fetching operands, one for use of the A L U, one for the memory operations cache **operations**, cache access and one for updating the destination register. And, while one instruction is being fetched and other instruction could be decoded and third instruction could be using the A L U, fourth instruction could be accessing the data cache and a fifth instruction might be updating its destination register. In other words there was substantial improvement in performance

because of the overlap of the execution of consecutive instructions; it is overlap in the sense that each of the instructions is using a different piece of the processor hardware.

Now, modern processors would actually be much more aggressive in their attempts to improve performance, by using more than just overlap. And, one of the directions which more almost all modern processors use is to exploit something known as instruction level parallelism. And, this is such a frequently used and prevalent concept that it is typically abbreviated as ILP; ILP standing for instruction level parallelism. And, the general idea of instruction level parallelism is that rather than just overlapping the execution of instructions, it should be possible to execute instructions in parallel with each other; if they are independent of each other.

In other words what we mean by in parallel is that two instructions might actually be fetched at exactly the same time, two other instructions might be being decoded at the same time, two instructions might be executing at the same time. And, this is we understand possible if the two instructions that I am talking about are actually completely independent of each other. So, this would clearly provide a much greater level of improvement in performance; since, instead of having five instructions occupying the processor hardware at a time, if I have two instructions being fetched, two being decoded, two being executed etcetera; I actually have ten instructions occupying the processor hardware. And, therefore, the rate at which instructions would complete execution could be substantially higher and this could lead to additional improvements in the performance of processors.

So, today we find that most processors in program in personal computers or laptops that you are using would be exploiting this kind of parallelism in their pipelines. And, they would therefore, we call ILP processors; processors that exploit instruction level parallelism.

(Refer Slide Time: 21:12)

The slide is titled "Instruction Level Parallelism Processors". It contains the following text:

- Challenge: identifying which instructions are independent
- Approach 1: build processor hardware to analyze and keep track of dependences
 - Superscalar processors

Below the text is a diagram of a pipeline. It shows five stages in boxes: IF, ID, EX, MEM, and WB. Above each stage is a small box containing a number: 1, 2, 3, 4, and 5 respectively. To the left of the IF stage, there is a vertical arrow labeled "pipeline width" with the number "2" next to it. Below the IF stage, there is a vertical arrow labeled "pipeline depth" with the number "4" next to it. A horizontal arrow points from left to right across the stages, with the text "N stage pipeline" written below it. A person is visible in the bottom right corner of the slide, appearing to be presenting.

Now, the challenge in the design of instruction level parallelism processor is being able to identify the instructions which can be executed in parallel. It is only if two instructions are independent of each other that they could conceivably be executed at the same time in parallel. And, there are two approaches that modern processors, the different kinds of modern processors used in order to undertake this activity. The first approach is to actually design the hardware, so that the hardware itself analyzes the independence between instructions. And, if there are dependence is it keeps to track of the dependences in order to make sure that instructions which are dependent do not move through the pipeline and cause the execution of the program to be in correct.

So, the management of the instruction level parallelism is done by the hardware in this first approach. And, the name given for these kinds of processors is to call them superscalar processors. The general idea that one could have in mind when thinking of the superscalar processors, is that until now we had a picture of a processor which might have a pipeline processor which might look something like this and they could be one instruction being fetched, one being decoded, one being executed, one using cache, one being return back. But in the case of a superscalar processor it is conceivable to try to think of the same processors as having, let say the capability of not only fetching one instruction but a fetching two instructions. So, I might think of it as a processor which has hardware complicated enough to fetch two instructions, complicated enough to

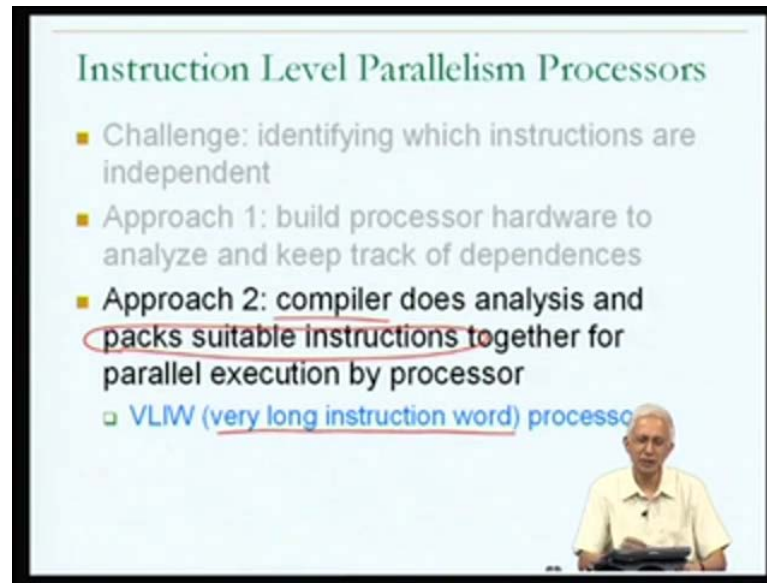
decode two instructions, to execute two instructions, to allow two instructions to be accessing cache at a time.

So, in some sense one could think of the superscalar processor has this hardware which is like the pipeline set that we saw before, but has this additional capability at each stage; so that more than one instruction can be processed in that stage and that will clearly require additional hardware. But this gives us the additional necessity to understand the processor with us having an additional dimension. Until now, we talked about the processor is having many pipeline stages, I talked about the N stage pipeline. And, we saw that N stage pipeline could conceive, in this case we have a 5 stage pipeline but in general we could have a 20 stage pipeline, where N is equal to 20. And, the speed up, the amount by which this pipeline could speed up the execution time of a program was related to N. So, this was one dimensional we talked about this as the pipeline depth; N was the pipeline depth a number of stages in the pipeline.

We now understand that in superscalar processors there is a additional dimension, which is the number of instructions they could be fetched or decoded or executed or using cache or return back at a time in parallel. So, in the example that I mention that it look like two instructions could be in each anyone of the stages at a time and that is not depth but what is referred to as the pipeline width; so, that is what is refer to as the pipeline width. And, in the example that I just mentioned the idea was the pipeline width was 2 and we could imagine that they could be pipelines which have a width of 4. In other words they have the capability of actually executing 4 instructions in one cycle or completing 4 instructions in every cycle.

And, you could understand that if there was a pipeline which are a width of 4, then you could conceivably has have 4 times the performance of a pipeline of width 1; pipeline of width 1 would be the simple kind of pipeline that we looked at in some detail. And, many of the processors that you may be you familiar with today have widths of 2, 3 or 4. So, this is one possibility as far as exploiting instruction level parallelism is concerned approach one; where the hardware is responsible for the management of dependences in parallelism and such processors being call superscalar processors.

(Refer Slide Time: 25:08)



The slide is titled "Instruction Level Parallelism Processors" in a green font. It contains a list of four bullet points:

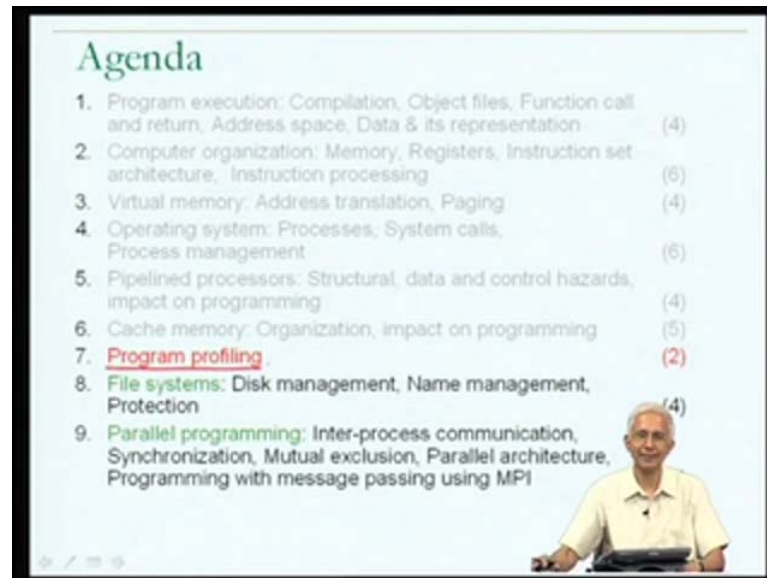
- Challenge: identifying which instructions are independent
- Approach 1: build processor hardware to analyze and keep track of dependences
- Approach 2: compiler does analysis and packs suitable instructions together for parallel execution by processor
- VLIW (very long instruction word) processor

In the bottom right corner of the slide, there is a small video inset showing a man with white hair, wearing a light-colored shirt, sitting at a desk and speaking.

The second possibility is that the job of identifying independent instructions could be done by the compiler. In other words the compiler could analyze the program before the program is executed. Identify instructions which are independent of each other and in effect pack those independent instructions together; so that they all of the independent instructions in a given cycle could be, if packed together could be moved through the pipeline together. And, this kind of an approach is what is known as the VLIW; standing for very long instruction word processor approach.

The term very long instruction word comes from this idea of actually packing many independent instructions into one mega instruction of some kind. And, consequently one could think of the larger instruction which is the packed version, if it has 4 mix one instructions packed into one instruction, to all of obviously have to be over larger size; hence the name. And, there are some commercial examples of VLIW processors as well. So, this is some suggesting in just as minor addition to what we have talked about in terms of pipelining; modern processors do more than the just simple kind of pipelining we talked about. With this I would like to wrap up discussion of the cache memory.

(Refer Slide Time: 26:23)



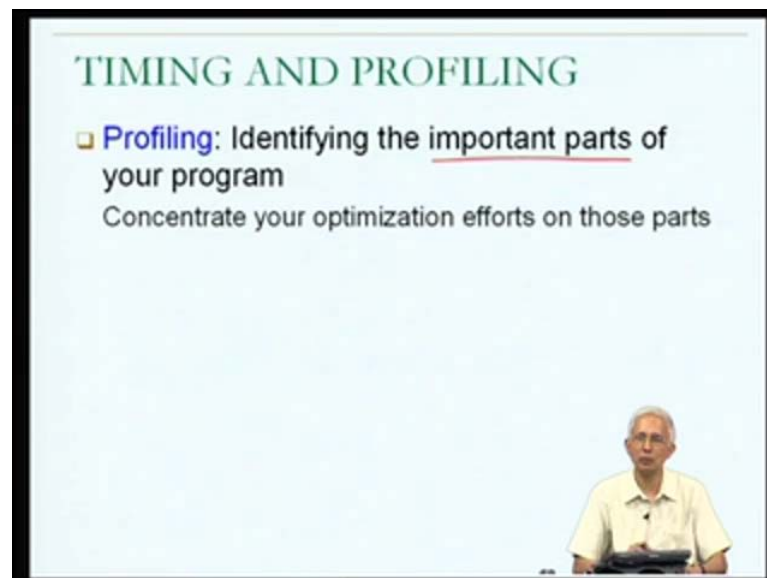
Agenda

1. Program execution: Compilation, Object files, Function call and return, Address space, Data & its representation (4)
2. Computer organization: Memory, Registers, Instruction set architecture, Instruction processing (6)
3. Virtual memory: Address translation, Paging (4)
4. Operating system: Processes, System calls, Process management (6)
5. Pipelined processors: Structural, data and control hazards, impact on programming (4)
6. Cache memory: Organization, impact on programming (5)
7. **Program profiling** (2)
8. **File systems**: Disk management, Name management, Protection (4)
9. **Parallel programming**: Inter-process communication, Synchronization, Mutual exclusion, Parallel architecture, Programming with message passing using MPI

A presenter is visible in the bottom right corner of the slide.

Our next topic, just going through the agenda sequentially refers to program profiling and without further adieu we will move into program profiling.

(Refer Slide Time: 26:35)



TIMING AND PROFILING

- ▣ **Profiling**: Identifying the important parts of your program

Concentrate your optimization efforts on those parts

A presenter is visible in the bottom right corner of the slide.

Now, in generally in this topic I am going to talk about two things; one is what I will call timing and the other is what I will call profiling. And, profiling is actually something we have been talking about before without using this term. And, it is essentially the activity of identifying the important parts of your program. And, what we mean by important parts; the parts of your program where the most time is spent, because it is these parts of

the program that you would want to look at more critically and trying to optimize your program.

For example, if had a **mechanism**, if I have a very large program thousands and thousands of lines in size, then you would clearly not be possible for me to do for example, pipeline analysis or cache analysis of the entire program. Because we saw that to do cache analysis, the number of hits, the number of misses that would be generated when the program executes going to some detail into the way that the different variables, their addresses associated with variables etcetera; interacted with each other. And, therefore, if one could on the other hand identify smaller regions of the program which are in fact significantly important, then one could concentrate once activity; optimization activity on the smaller regions of the program and get possibly a very good benefit in terms of proving the execution time of the entire program.

So, the activity of identifying the important parts of your program from the perspective of wanting to spend your optimization efforts on just those important parts of is what is typically refer to as profiling. And, it is obviously going to be important to us; we have understood a lot about the hardware and the software side of what happens when a program executes. But in order to do something about improving the performance of our program, we may have to know which part we should concentrate on. So, for this is a critical part of high performance programming.

(Refer Slide Time: 28:22)

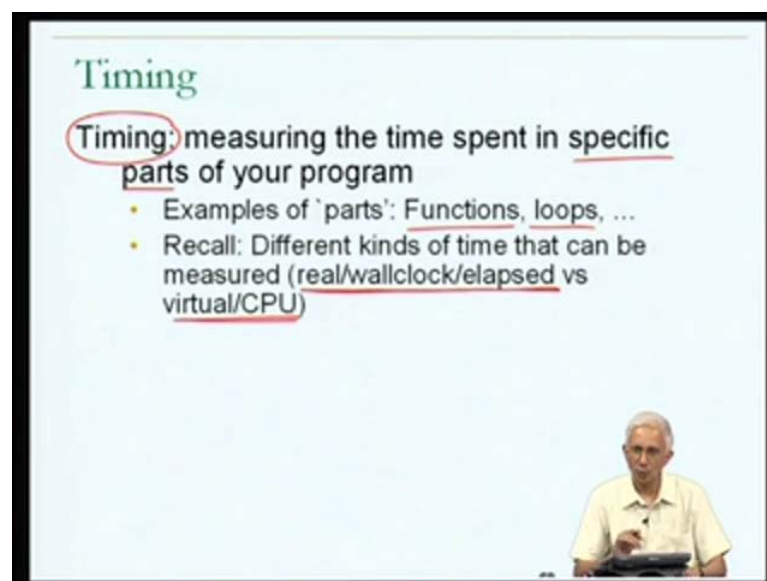
TIMING AND PROFILING

- ▣ **Profiling:** Identifying the important parts of your program
Concentrate your optimization efforts on those parts
- ▣ **Timing:** Determining program execution time

Now, you would guess that an important part of doing profiling might be having the capability of getting time estimate; so, the knowing the amount of time that it takes so your program to do something. And, timing is in general the term given for determining program execution time, but I could also referred to timing as activity of identifying the amount of time spent in some part of my program. For example, if I have identified a particular function of my important as being important, I may want to know how much time was spent just in that function, I may not be too concerned about total program execution time on the total program execution time; I might be more interested in getting the time as far as some important part of the program is concerned. Therefore, associated with profiling they may be the need to know something about timing.

And, hence you will spend one and half lectures or two lectures on understanding concepts and capabilities for doing things like this, timing and profiling of our programs.

(Refer Slide Time: 29:18)



Let me start with timing. Now, the objective here is to measure the time spent in specific parts of my program and could be that I want to know the time spent in next feeding the entire program, it could be that I want to find get an idea of the time spent in one of the important parts of my program; such as one our particular function or a particular loop and what could I mean by the different parts of a program. Well, we have seen examples where they were loops that we analyzed.

So loop; a loop like a dack speed loop or a vector sum loop etcetera, could be examples of loops. The other examples that I used were, I might be interested in trying to improve one particular function of my program. So, in general when I say parts, I could be talking about functions or loops. And, at this point I will just remind you that when we talked about time earlier in this course, we distinguish between a few different kinds of time. Because we understood that on a machine where there is only one C P U, the operating system is creating the elusion of many programs being able to share the C P U. By actually sharing the C P U time among the different processes used that happen to be in memory. And, therefore, there was a need for us to distinguish between at least two different concepts of time; one must the actual elapsed time as we would see an a clock on the wall and that is refer to as the real wall clock or a elapsed time.

And, the other is this virtual process related time; the amount of time or of the C P U that a particular process are actually got and it is it should be possible for us to find both of these times. And, the mechanisms that we are going to look at will either have, will have to identify as providing you and estimate either of the real time over of the virtual time. And, unless you are aware of what a particular mechanism is giving you, the time itself may not be of much value to you in figuring out how your program is behaving.

(Refer Slide Time: 31:15)

Timing

Timing: measuring the time spent in specific parts of your program

- Examples of 'parts': Functions, loops, ...
- Recall: Different kinds of time that can be measured (real/wallclock/elapsed vs virtual/CPU)

1. **Decide**
 - [which time you are interested in measuring
 - [at what granularity
2. **Find out what mechanisms are available and their granularity of measurement**

Therefore, in doing a timing estimate, but in doing timing experiments some timing runs of programs that you are interested in; there are few decisions that you have to make.

The first decision that you have to make is, what are, what time are you interested in measuring? Are you interested in measuring the real wall clock or a lapse time or are you interested in measuring the virtual C P U time? And, they may be some situations where you interested in the first, in some situations where you are interested in the second. So, this is the first decision that you have to make. And, secondly another decision that you have to make is, how fine an estimate of the time do you need? Would it be in, for example, would it be enough if you got an estimate that was accurate to the nearest second? Or do you need the, an estimate that is accurate to the nearest clock cycle, nanosecond? So, that is another decision that you have to make.

And, you will clearly understand that the decision has to be made in out of for you to use the correct mechanism for measuring time, different mechanisms will be provided. Some of them might be capable of riding you elapse time only accurate to second, others might be capable of giving you C P U time accurate to the nanosecond, but you have to correct pick the correct mechanism depending on what you are objective is. So, if you first have to decide what your objective is; once you have decided what your objective is, you look for the correct timing mechanism which satisfies your objective. In other words if you are looking for a elapse time you look for mechanism that can give you elapse time measurements and of the appropriate accurate granularity.

(Refer Slide Time: 32:42)

```
time command % a.out
Usage: % time a.out
Example: % time ls
0.00user 0.002sys 0:0.003elapsed
```

Handwritten annotations in red:

- Under `ls` in the example: `ls`
- Under `0.00user`: `virtual CPU`
- Under `0:0.003elapsed`: `wall clock real elapsed`
- Under `% time a.out`: `% a.out`

A small inset image of a man in a white shirt is visible in the bottom right corner of the slide.

Now, let us just look at a few of the time relative mechanisms that you will find on Linux or Unix systems. Now, one mechanism which some of you may have come across and may even have used is what I will refer to as the time command. And, you may have in fact used it and the way that you use it is that in response to the shell prompt. If your intention is to find out the amount of time that it takes in the execution of let say the program a dot out, then rather than just executing a dot out which you would have done like this; you execute a dot out through the time command, in other words you type time space a dot out. And, now the time command will cause a dot out to be executed but it will give you or in return information about how much time you took for a dot out to execute.

Let me just give you a few examples. So, let us suppose that in response to the shell prompt, I typed time l s. Those of you have used Linux or Unix systems, will recognize l s as a command which can be used together with the listing of the files within a directory, we learn more about files and directories a little later. But this is l s is a command which gives you some information about the different files which are available for access. Now, when you type time space l s, obviously the l s command will print out the information that it is supposed to I am not going to show you the output of the l s command; I am just going to show you the possible output of the time command. I believe this was taken on a Unix system, this particular output.

So, you will notice that the time command is giving me three pieces of information, one is labeled as user, one is labeled as system and one is labeled as elapsed. And, therefore, we would understand that the time command is giving us the elapsed time but in addition it is giving us C P U time or a virtual time. So, the elapsed time is obviously the elapsed or real or wall clock time, that we talked about earlier. Whereas the other two times which have this breakup into user and system what we would refer to as the virtual of the C P U time.

So, the time command is giving us both; it tells us both how much time was spent in terms of the virtual timeline of this particular process in user mode and that number was so small that it registers as 0, as well as the amount of time that was spent in the execution of l s in system mode and that is given as 0.002, I believe this the unit here is seconds. In addition to this it gives us the amount of wall clock time the actual elapsed time. The amount of time I would have observed if I had time, if I had looked at a wall

clock before and after the time command was executed and that is given as 0.003. So, in this particular case we understand that the bulk of the time of execution of the ls command was spent in system mode very little, practically no time was spent in user mode. And, the bulk of the elapsed time and the relationship between the elapsed time and the amount of time spent to the virtual mode is fairly close.

(Refer Slide Time: 35:46)

```
time command
Usage: % time a.out
Example: % time ls
0.00user 0.002sys 0:0.003elapsed
Example: % time man csh
0.268user 0.032sys 0:15.486elapsed
```

Reports Real/Elapsed/Wallclock time, CPU time in user mode, CPU time in system mode

Now, let us just look at another example, this is a more complicated example. Here I am going to time; I am going to use the time command to find how much time it takes to execute the command man csh. Now, what is man csh? We have come across csh before; I could have run the same command type by typing man c s h. We have come across csh before, csh was one of the shells that is available on many Unix and Linux systems; it is a command interpreter. What about man? Well, man is the common in Unix and Linux systems as command that can be used to read a manual entry; so, man stands for the manual or the book which contains information about the different commands and that is available on line.

So, when I type man csh in effect what I get is a listing on the screen, a printout on the screen of the manual entry for the csh command. And, this is a very long manual entry and typically when you type man csh, you do it with a purpose of reading some section of the manual entry and therefore, you suspect that if somebody did a man c s h, they would spend some time reading the page as it is on the screen. How will that reflect itself

in the output from the time command? If you look at the output from the time command and once again, I am not showing the output from the man csh because that would be a 30 or 40 page document; so, we do not want to look at that on screen. But you will notice that the user time is no longer 0; there is a significant part over second spent in user time. This is small amount of time compare to this which is spent in a system mode and the fairly large amount of time that is spent in actuality, if I look at the elapsed time.

So, I believe this is 15 seconds of time of which 0.032 seconds are spent in system mode and 0.268 seconds are spent in user mode. So, if I look at the sum of these two times it adds up to about 0.3 seconds. And, I look at the amount of that was actually spent in a elapsed time its 15.48 seconds. In other words I spent 15 seconds reading the manual entry and the actual amount of C P U time, both in user mode and in system mode that was spent on this command was only a third of a second. So, where was the remaining 15 plus seconds spent? It was obviously spent, because I as a user was reading this information of the screen in order to understand something about the use of csh and that is reflected in the elapse time, but not reflected in the user time in the virtual C P U time.

Because during the time that I am not interacting with the program, the process itself might in fact not be running; it might be innovating mode, it might get blocked. So, this is deviling to us and gives us some idea about what kind of time we should be looking for depending on our requirement. So, in effect the summary is that; this the time command reports both real elapsed wall clock time as well as C P U time or virtual time. And, as far as the C P U time is concerned it gives you a breakup between user mode and system mode.

(Refer Slide Time: 38:55)

```
gettimeofday()

Reports real time that has elapsed since 00:00 GMT 1 January 1970 (The Epoch)

#include <sys/time.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);

struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};

Usage: Insert calls to gettimeofday in your C program
```

Now, another command which might be useful to us is called `gettimeofday` and I am using this notation because `gettimeofday` will be invoked like a function from within a program. So, in order to understand how to use `gettimeofday`, I told you about the usage of the `time` command by just showing you how you could use the `time` command from the shell prompt. The `gettimeofday` we are, we have told this a function which means that we will be using `gettimeofday` from within our program. You will write a program which can call `gettimeofday`. And, therefore, I tell you a little bit here about the `gettimeofday` function; it has some parameters and the details about the structures used in the parameters as well as the function itself are available in an include file.

So, if you want to use `gettimeofday` you would have to include this include file in the program. Now, if you actually looked at the `timeval` struct and you would look at it inside the `sys/time.h` include file, you would see that the `timeval` struct contains two fields. One of which is a field that report seconds and the other is a field that is label microseconds. The fields are called `tv_sec` and `tv_usec`. So, clearly this is structure which contains an **estimate**, something to do a time to the granularity of microsecond apparently. And, the question is how do we use this? In order to use it we have to know what it the meaning of the value that it returns is; and this information I am giving you in that yellow block up at the top.

So, what `gettimeofday` reports, what is a return as its returned value? In the struct `timeval` parameter is, the real time that has elapsed since 0 0 0 0 Greenwich mean time on the 1 st of January 1970 and the manual entries that you read in connection with `gettimeofday`, we will refer to this instant in time as the epoch. So, basically what this is returning to us is a number of seconds that have elapsed, since 0 0 0 0 Greenwich mean time on the 1 st of January 1970.

Now, that is rather curious thing to return. But you will understand that if I call `gettimeofday` now, I will get the number of seconds that have elapsed since 1 st January 1970 and then if I wait sometime and I call `gettimeofday` again, then I will get the amount of time that has elapsed once again since 1 st January 1970. And, the differences between these two return values will tell me how much time has been spent by my program in between. And, that I can therefore, use the `gettimeofday` to estimate the amount of time spent in a part of my program. So, the idea is that I could use `gettimeofday` by inserting calls to `gettimeofday` in my C program before and after the regions of interest to me, in terms of the region particular region of interest to me in my program; which I want to know how much time I was spent in execution.

(Refer Slide Time: 42:00)

Using `gettimeofday()`

```
struct timeval before, after;  
gettimeofday(&before);  
[ region of program you want to time  
gettimeofday(&after);  
printf ("%d\n", after.tv_sec - before.tv_sec);
```

Your C program

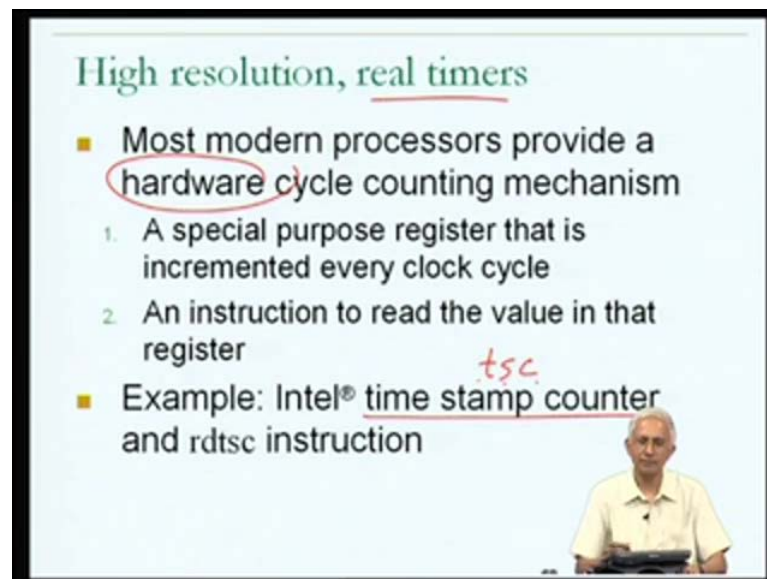
real elapsed time wallclock

So, let us consider a C program and let suppose that they have some particular region that I am interested in, I am interested in this region I want to know how much time was spent execute in this part of my program. So, what I can do is, I can setup two `timeval`

structures; one called before and one called after. I can call `gettimeofday` passing the parameter before, before the region of interest to me and call `gettimeofday` passing the parameter after, after the region of interest to me. Subsequently, I can compute the difference between the `sec` fields, the second fields of the after structure and the before structure, to find out how much time was spent in that particular region of my program. And, I could print that out using a `printf` statement in order to actually get the output; which I could subsequently look at to understand, how much time was spent in this region of my program.

Therefore, you can use `gettimeofday` by inserting calls before and after and subsequently computing the difference between the after value and the before value; in order to measure the amount of real time. So, the important thing to remember is that we are, what we are talking about here is *elapsed time*; `gettimeofday` returns, a real or *elapsed time* or wall clock time value.

(Refer Slide Time: 43:17)



The slide is titled "High resolution, real timers". It contains the following content:

- Most modern processors provide a hardware cycle counting mechanism
 1. A special purpose register that is incremented every clock cycle
 2. An instruction to read the value in that register
- Example: Intel® ^{tsc} time stamp counter and `rdtsc` instruction

In the bottom right corner of the slide, there is a small video inset showing a man in a light-colored shirt sitting at a desk.

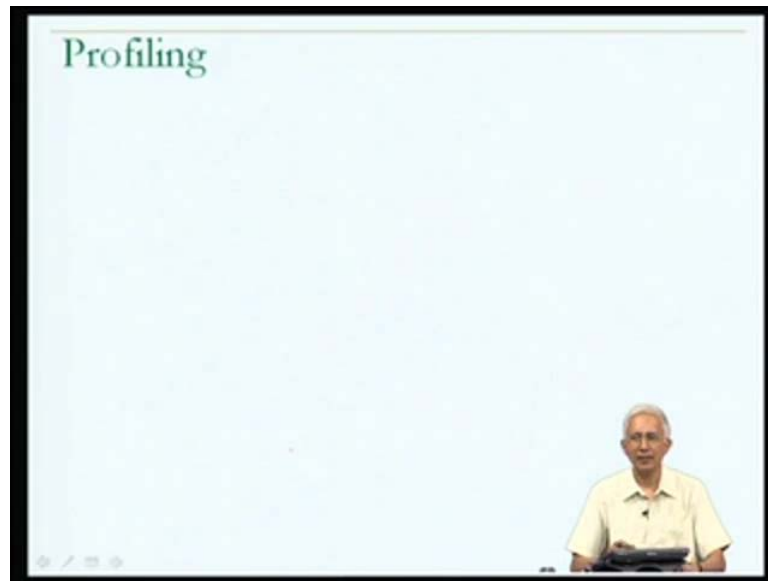
Now, the next question is whether there are other timers which can give us good estimates as far as high resolution or very nanosecond type of estimates of how much time are being taken by in parts of our programs. And, just to quickly give you an idea about this, I like to mention that almost all modern processors provide a mechanism which can be used for this purpose. And, these mechanisms are hardware, they are built into the hardware and they basically return an account of the number of cycles that have elapsed.

And that therefore, one could use a hardware cycle counting mechanism. And, if one knows how much time is spent in one cycle of that processor; for example, if I know that this is a 1 gigahertz processor, then I know that one cycle is equal to 1 nanosecond; then by counting the number of cycles or by using a cycle counting hardware mechanism, I can get very accurate real estimates of real as in real elapsed wall clock estimates of how much time was spent in a part of my program. Now, how would these mechanisms be made available to the programmer, I have suggest that they are available in the hardware.

Now, if such a mechanism is available in the hardware, we would suspect that in the C P U would contain a special purpose register that is incremented every clock cycle. And, that is use by referring to this register that the cycle counting mechanism works. So, if the processor has this register and the register is incremented every clock cycle, then as long as there is an instruction using which we can read the value of that register and printed out; then we can use this mechanism in our programs. So, in addition to the special purpose register there is incremented every clock cycle, all that the hardware has to provide is an instruction to read the value that is in that register. And, this is as I said available in almost all modern processors.

For example, the Intel processors have something called the time stamp counter; they do not call at the cycle counter they call at the time stamp counter or t s c time stamp counter. And, there is an r d or read t s c instruction; r d t s c instruction using which one can access the time stamp counter. This is not a privilege instruction, is an ordinary user mode instruction. And, almost all processors today have similar mechanisms. So, one could use this instruction before and after the region of interest in the program and by calculating the difference between the two and knowing the amount of time that a cycle is, one can give very accurate estimates of how much time is spent in a region of the program.

(Refer Slide Time: 45:38)



Now, with this idea about timing, we understand that there are mechanisms at the level of full program such as time, there are mechanisms at the level of parts of a program such as `gettimeofday`. And, there are even more accurate mechanisms which will give a cycle counts of parts of a program such as the hardware time stamp counter type mechanisms. Now, let us go back to the primary question of interest to us that relating to profiling; how do we find out the important parts, how do we identify the important parts of our program? Now, in order to do this we could of course, use the timing mechanisms that we have seen, surrounding the different suspected important regions of our program by calls to `gettimeofday` and calculating the differences. We could do this for all the regions for our program and then identify the important regions by such a mechanism.

But this would be a mechanical process for which we would have to include the calls to `gettimeofday` at various points in our program and might be an inconvenience for extremely large programs. So, what we would prefer instead in some kind of a mechanism which does this automatically for us. In other words we would like to have access to a tool which does a profiling for us. And, if there was such a tool, it would obviously be called a profiler.

(Refer Slide Time: 46:48)

Profiling

- **Profiler:** A tool that helps you identify the 'important' parts of your program to concentrate your optimization efforts
- **Profile:** a breakup (of execution time) across the different parts of the program
- Can be done by adding statements to your program (**instrumentation**) -- so that during execution, data is gathered, outputted and possibly processed later
- **Automation:** where a profiling tool adds those instructions into your program for you

39

So, going back to the slide a profiler would be a tool that helps you identify the important parts of your program; so that you can concentrate your optimization efforts on those important parts of your program; and there unfortunately many different profiling tools that are available. And, the word profile is used to refer to a breakup typically of execution time across the different parts of the program. So, you use a profiler to get a profile of your program; the profile may tell you how much time is spent in each of the functions of your program or each of the loops of your program. And, using that profile you can quickly identify what are the important functions or the important loops of your program.

Now, as I had mentioned earlier you could do this on your own, by adding statements into your program such as calls to `gettimeofday`. But an alternative is to get this done for you by a profiler to save you yourself the new sense of having to do this insertion of additional statements into your program. The insertion of those additional statements in the program is what might be referred to as instrumentation. Instrumentation itself is a discipline, it is a branch of engineering; what one studies in instrumentation is the construction of instruments. Instruments are devices which can be used to make a measurement. And, if you think about it the profiler that we are talking about here is in fact a tool which is being used to make measurements; measurements through which we identify the important parts of our program and the way that the profiler could work is by adding statements into a program.

And, therefore, the statements which are added are the instruments through which the profile is obtained and hence the process of adding those additional statements is refer to as instrumentation. So, we are currently trying to understand something about the profilers which will do this for us automatically; so, we do not have to add the instrumentation by ourselves.

(Refer Slide Time: 48:36)

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)

Now, there are several different profiling mechanisms that are available on the typical system. And, these profiling mechanisms will differ in a few ways; one way that they will differ is in the level of granularity that is supported. And, to some extent we can think of the granularity as being the size of the important region, that we are talking about. I talked about using profiling to identify the important parts of our program; so how big is a part that is what we are talking about in the section about level of granularity, so that a some profilers which give you information at the level of functions. So, the level of granularity is the function available and they would be called function level profilers. There are some profilers which give you information at the level of statements call for example, lines of a program and they may be called statement level profilers.

There are yet other profilers which give information at other levels; one of the more popular kinds of levels of profiling is something called to the basic block level. And, roughly a basic block is a sequence of a contiguous instruction in a program with a

property that; that sequence of contiguous instructions in the program has a single entry point which is the first instruction in the basic block and the single exit point which is the last instruction in the basic block. So, in some sense the basic block is some series of instructions in a program such that you enter the basic block only from the first instruction and you leave the basic block only from the last instruction.

In other words if a particular basic block; a basic block is a series of instructions. So, it might start with a load word instruction then they might be in add instruction and so on. But we as far as a basic block is concerned, we **get** from the by the definition, we understand that if the first instruction in the basic block is executed, then the all instructions in the basic block will be executed exactly once. So, that is why this is a nice way to look at a program, one could rather than thinking of a program as being made up of functions or just statements; at a lower level one could think of the program is being made up of basic blocks. And, hence some people like to use basic block level profilers. So, profiling mechanisms might be available at different levels in the light of the granularity, the size of the important parts that could be identified.

(Refer Slide Time: 50:56)

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 mechanisms at the function and basic block level**

11

And, in general profilers may also differ in the nature of the kind of data that they give you. Until now I was assuming, we were assuming that we were interested in finding the amount of time that was spent in executing an important part of our program. But they could be other considerations that are important other than time. For example, one

clearly, one important kind of profile data might be a breakup of the amount of execution time spent in the different parts of my program or how much time was spent in a particular function.

But in addition to this it might be adequate, if I knew how many times each function was executed. If for example, I knew that particular function `(())` was executed 30,000 times and all the other functions in the program were executed only 1 or 2 times. Then that would be possibly enough reason for me to try to concentrate my attention on that function called `(())`. So, even just a count of how many times a function or a statement or a basic block was executed might be a value to us. And, that would be a different kinds of profile data; rather than getting a breakup of the amount of the number of seconds spent in each function, I may just get it feedback on the number of times each function was executed and that might be different kind of profiling mechanism.

So, there are will be different kinds of profilers, they will defer in their level of definition of important part of program. And, they could differ in the kind of information that they provide in terms of trying to classify something is important; they could be based on times they could be based on counts.

Now, what we are going to do is we are actually going to look at two specific profiling mechanisms; we are going to look at exactly how they work, we are going to look at one of the mechanisms, we are going to look at is at the function level. The other mechanism we are going to look at is that the basic block level. We will understand the mechanisms that I used behind the scenes, we will understand. Therefore, the guidelines which should be taken into account in trying to use these mechanisms. And, we will proceed by looking at the function level profiling mechanism in the next lecture, we stop here for today. Thank you.