Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 30
The Memory Systems Part-I

Welcome to chapter 10. In this chapter we will discuss the memory system. So, up till now we have been very vague about the memories that are used in typical processor system. So, in this chapter we will look into this in great detail. This is again the tenth chapter of the book computer organization and architecture. It has been published by Mc Graw Hill 2015, and you will get copies of this book in almost all the countries via amazon. If there is any problem with the availability you can kindly send the author or the publisher an email.

(Refer Slide Time: 01:42)

So, we will discuss this chapter we will discuss 4 separate sections. One is the overview of the memory system, then we will discuss caches, we will then go into the details of a memory system look at mathematical models. And finally discuss something called virtual memory.

(Refer Slide Time: 02:04)

So, up till now we have assumed that the memory is one large array bytes. We have assumed that the memory is one large array of bytes. This large array starts at 0 and ends at 2 to the power 32 minus 1, if we assume 2-bit memory system. So, every program will perceive the memory as one huge array of bytes, where each of this is 1 bytes, and each byte has an address that is the memory address. See takes we have also seen that it takes one cycle to access the memory which means the form of read or write. So, this is the assumption that we have been making that when we write a program in assembly program to be specific, the entire memory in the system is assumed to belong to that program, and the program can write any part write through any part of the memory space at well. So that is not what happens in reality. What happens in reality is something like this?

We have many programs running at the same time, and we have to somehow magically avoid overlaps between programs running on the same processors right and running on the same processor. Also what we have assumed is that all our programs require less than 4 GB of your space which again is might not be true. So, for example, we have a processor. So, we have a processor over here, and the memory might be 1 GB, 1 gigabyte. So, in this case we have to live with this limitation smaller program still need to run over here. And it is also possible that one of our programs are you know we can have let us say 64 bits addressing. And it is possible that we have a programming that requires 8 GB of space and we have 1 GB of physical memory. So, how would we run

such kind of a system? This is what we will discuss in the latter half of this chapter, when we will discuss virtual memory

(Refer Slide Time: 04:50)

So, let me try to motivate this as follows. So we have essentially made 2 assumptions. One assumption that we have made is that, one program you know one running program in a sense owns the entire memory system. So, that is one first assumption that we made that the entire memory system belongs to one program right. So, basically that is the first one, the other assumption that we have made is that for a 32-bit memory system the physical memory is 4 gigabytes. That need not be the case.

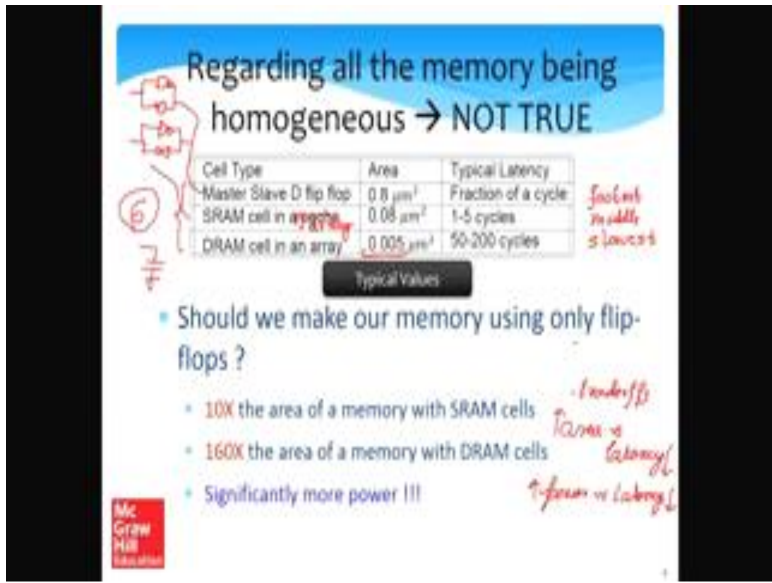
So, we can have a small physical memory for example, we can have maybe a one gigabyte physical memory. And then you know we need to see how to run a program on such kind of a system. So, let us first try to look at the assumption number one. So, this is the screen capture of the windows task manager. So, the windows task manager shows all the programs that are running on my system at this very point of time. And as you can see I am running a skype which is taking 89 megabytes of memory. I am running power point which is taking around 79.8 megabytes of memory. Then I am running many more programs as you can see if I just scroll down, I may be running a 100 odd programs. Sorry I am running 74 I am running 74 programs. So, process is actually running instance of a program means I am running 74 processes.

So, it is not that my you know my CPU is running the 74 at the same time, it is of course, it runs one program at a time, but it will switch between programs periodically. So, let us say you know it is running program number one then the CPU will switch and it will run program number 2. Then it will switch and it will run program number 3. So, it periodically switches right between programs and again it can you know again switch back. So, when I am running all of these programs and each program is assuming that the entire memory space belongs to it.

There is a possibility of an overlap. In the sense that one writes to one memory address another program is taken writes to the same memory address. So, we have an overlap. So, this is something that should be avoided, but the most important point that should come out from this slide, is that at any point in a system my multiple programs running you know are alive at the same time, but of course, the given processors, let us say there is one processor, one processor can only run one program at one time right.

So, if it is running only we are running multiple programs or it runs one program and after sometime, it again it switches and it runs one more program. Right again it is an it is sort of switches between the programs, but even if it switches we have to ensure that in a one program has returned to so, let us assume that this is the memory space that we have. If one program returned to you know these regions of the memory space, then other program does not the next program does not touch the spaces. Otherwise there will be a problem.

(Refer Slide Time: 08:51)



So, we need to solve this. Also we have been making some more assumptions we have been assuming that the entire memory is you know it takes the same time. It takes one cycle to access any part of the memory system. So, that is not correct. So, let us take a look at the different kind of technologies that we have learnt, which can actually be used to make a memory system. So, one is a master slave D flip flop that we talked about. So, it is area you know typically is very large. So, it is around 0.8-micron square. So, it is a fairly large structure, but it is fairly large and fairly fast as well. So, in a fraction of a clock cycle it is possible to access the flip flop and also do more work.

In contrast if I consider an SRAM cell you know in a cash or in so, I should rather make it in an array of SRAM cells. So, if I consider it as a single SRAM cell in an array for storing a single bit, it is actually 10 times more area efficient. So, the area that it takes is around 0.08 micron square, and the typical latency will be 1 to 5 clock cycles; if I consider a DRAM cell in an array. So, it is even more area efficient. So, it is even more area efficient in SRAM cell. So, it only takes 0.005-micron square. So, these are you know slightly old values, but the ratios will still remain the same if I consider the current technology. So, DRAM cell would take something similar 0.005-micron square. And the typical latency; however, for accessing a DRAM array is very high. It is around 50 to 200 cycles we can see that the DRAM is the slowest and D flip flop is the fastest.

Right, but well speed comes at a price. And the price is area efficiency and this is sort of at the middle right. So, SRAM is in the middle. So, as we see that as we increase the area and increase area also means increase power, the latency decreases; that means, our cells are faster. And similarly when we go and when we use the DRAM cell a DRAM cell the area is very small, but the latency is high. So, all of these things you can again go back to chapter number 6 on memories and you know DRAM cells and SRAM cells.

In case if some of you have forgotten you can go back to chapter 6 and look at it once again. So, just to refresh your memory a DRAM cell is actually a single capacitor. A SRAM cell is a cross coupled inverter. And a master slave D flip flop well that essentially consists of cross coupled nand gates. You know 2 of these basically one of this and one more some additional complexity. So, this is there in chapter 6, but the basic idea is certain trade off exists between latency and area. So, should we make our memory only using flip flops well it is a very bad idea, because for a given amount of chip area will be able to fit only very few bits.

In contrast, so, rights are 10 times area of a memory with SRAM cells roughly and 160 times area of a memory with DRAM cells. So, you know these are represented in numbers. And also they will consume significantly more power. So, we cannot use in a any single technology to make the memory right. That is the most important take home point that is coming from this slide, that to make a memory systems there are tradeoffs. So, maybe I can write down. So, there is area versus latency tradeoffs. So, as the area increases, the latency decreases; and similarly the power versus latency tradeoff as well. So, as the power increases we have more power the latency decreases; however, you know the entire memory cannot be made up of just flip flop cells. It will not contain it is capacity will be very low. It will not be power efficient it will take consume too much power.

Similarly, the entire memory cannot be made of just DRAM cells; because we can fit in a lot of bits it will be too slow.

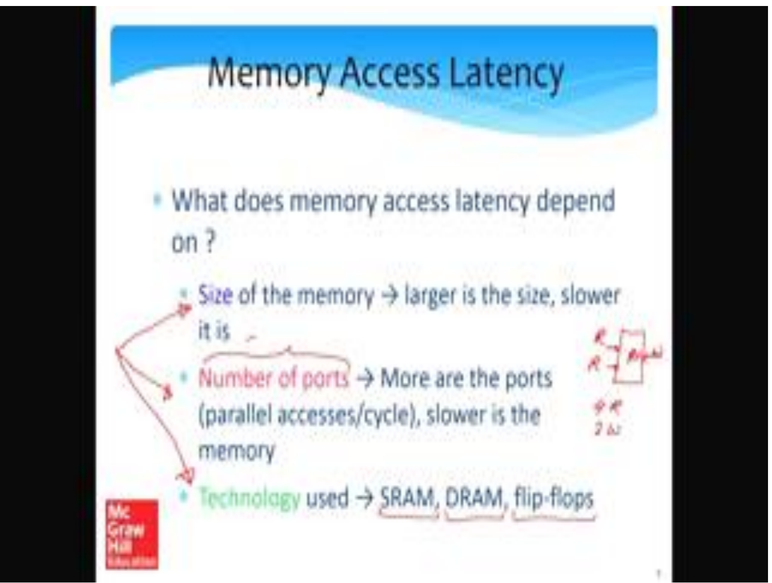
(Refer Slide Time: 14:34)

So, given these things we can nicely summarize all the tradeoffs over here, something that we discussed in the previous slide. If I increase the area I will reduce the latency of course, but increase the power. If I make the memories as faster I will increase the area, but I will increase the power as well. And if I reduce the power well I have to make them smaller and I have to make them slower as well. So, you cannot have best of all worlds. So, it is you know it is a philosophical thing that is true. That having the best of all worlds is not possible right. That is simply not possible.

(Refer Slide Time: 15:26)

So, what is in a sense desirable is that, we have something some kind of a solution which is a compromise. So, you know as we have been discussing having a memory with just flip flops we will not be able to store anything. We just SRAM cells well we need more storage and that will also will not do and the DRAM cells will have a lot of storage space, but every access will be very slow.

(Refer Slide Time: 15:47)



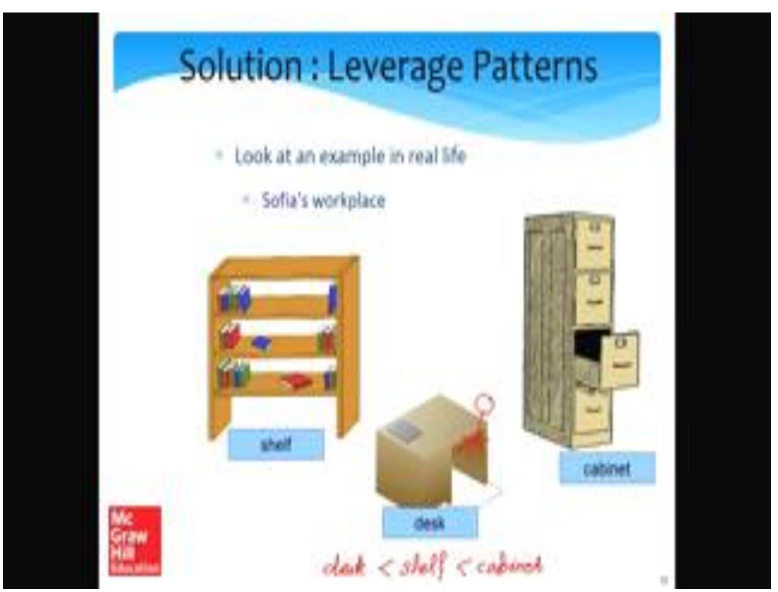
So, what will we do? So, memory latency depends on well the size of the memory yes. So, larger is the size lower it is. The memory access latency also depends on the number of ports. So, this is let me discuss this. See consider a small memory like a register file. So, in a register file there is one instruction which is reading in one cycle and there is also one instruction is writing. So, the instruction is reading 2 registers in the same cycles you need to read ports 2 interfaces to read. Similarly, one instruction is writing you need one right port. A port is essentially an interface to write. So, to read and one write right.

So, we can always have a slightly bigger we can always have a different kind of processor, that instead of one instruction issues 2 instructions per cycle. So, what would this processor require you need 4 read ports and 2 write ports 6 ports right. So, we have one more so more are the number of ports number of parallel in accesses per cycle slower will be the memory and slower also will be you know more will be the power. And then the latency also depends on what kind of technology we use. If it is a SRAM or

a DRAM or a flip flop it will take you know different amounts of power and latency in areas as we have discussed.

So, what are the main things that are in our control one is the size right. So, we can tweak with the size of the memory. So, if let us say the storage capacity is low. So, if it is the 4 kilobyte memory will be very fast, here it is the 4 megabyte will be fairly slow. And the numbers of parallel accesses we are supporting right the number of ports and the technology that we are using.

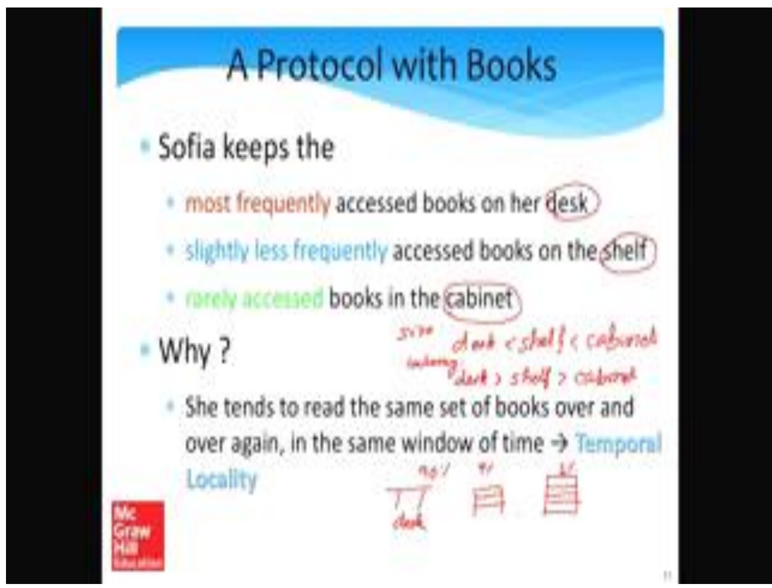
(Refer Slide Time: 17:51)



What are we building it with SRAMs or DRAMs are flip flops? So, let us look at a solution, but let us first search for a solution in real life. So, let us leverage some pattern. So, let us consider a student Sofia’s work place. So, let us assume that Sofia is sitting over here right on a desk. And she has some books on the desk. And nearby she has a shelf which has some more books. And there is a cabinet there is a huge cabinet that is faraway right clear clearly the desk is small.

So, if I in terms of size the desk is the smallest, and this is smaller than the shelf, which is smaller than the cabinet. Let us say the cabinet is huge cabinet is huge faraway. So, what is one thing that we can see from here? Accessing a book from the desk is fast and quick, but the desk also has small size. Then we have the shelf which is slightly larger and it can fit more books and then where the cabinet which is really large.

(Refer Slide Time: 19:13)



So, what would be the right way of all for Sofia to organize her books? The right way will be for Sofia to keep the most frequently accessed books on her desk now it is you know the books that she is reading for example, if she is preparing for an exam she can keep the books that are the most related to her course to her exam on her desk. She can keep the slightly less frequently accessed books on the shelf right. You know some books some reference books can be kept on the shelf which are not used all the time, but you know can be used sometime. And the rarely accessed books which she hardly ever reads there will be many such books and they can be kept in the cabinet right. So, what is more important is we need to note these 3 words that desk a shelf and a cabinet, a desk is small and the it is essentially a desk is small and fast. So, what this means is that you can get a book from the desk very quickly.

In a contrast a cabinet is large and slow. So, it takes a lot of time to actually find a book inside a cabinet because it is large. So, that is the reason it is large and slow, but it has more capacity. And a shelf is somewhere in the middle. So, if I will just look at you know size. So, basically I just can write it once again, that a desk is smaller than a shelf is smaller than a cabinet. If I look at latency or time it takes to get a book fetch a book, then it is reverse a desk is faster than getting a book from a shelf, which in turn it is faster get a book from the cabinet.

So, why does this strategy makes sense? Well strategy makes sense mainly because of nature of human behavior right. So, let us say before preparing for an exam she will tend to read the books which are on the desk more and more right over and over again she will read the same books because she is preparing for exam and so in the same window of time which is maybe a day or 2 or 3 days before an exam, a certain set of books will be accessed. After that if there is another exam one more set of books will be accessed. So, any kind of such a pattern is called temporal locality. Temporal means time it is the adjective form of time.

So, in a short duration of time when we tend when we access the same thing over and over again, that is called temporal locality in this case before an exam the same books will be accessed over and over again. Because of the phenomenon of temporal locality, it makes sense to have a small desk where we can keep our books. So, most of the time may be you know 90 percent of the time we get the books from the desk. Then again it makes sense to have a shelf and out of the remaining 10 percent right maybe you know 9 percentage of the time, we can get the books from the shelf, because the shelf is larger. And then again we can you know have a large cabinet where the remaining 1 percent of the time we need to go.

So, the advantage of this particular organization is that 9 in 10 times we can get the books within arm's reach. Out of that 9 in 100 times, we can get the books from the shelves and maybe Sofia just needs to walk 10 steps. And only 1 in 100 times, there she actually has to make a trip to the cabinet which is at the end of the room. So, it sounds reasonable this is typically the way that we organize things. So, even on a kitchen which is actually what we do; on a kitchen counter top, we have we have very commonly used ingredients right. For example, salt, sugar that is there on the counter top.

Then we have a small shelf or we have a refrigerator, which contains other items which cannot be kept on the counter tops because of space. So, that includes vegetables and meat. And then we might have some more items that we use very rarely. So, in that case it does not makes sense to keep them at home, we can always you know if we want to make something that uses a very rare ingredient like saffron maybe, we can go to the market and buy one. So, this is similar.

So, this pattern is again temporal locality in action, where what we are doing is that we are basically looking at all our accesses and trying to derive a pattern of it. So, be it accessing memory or be it anything else, we always have some amount of temporal locality.

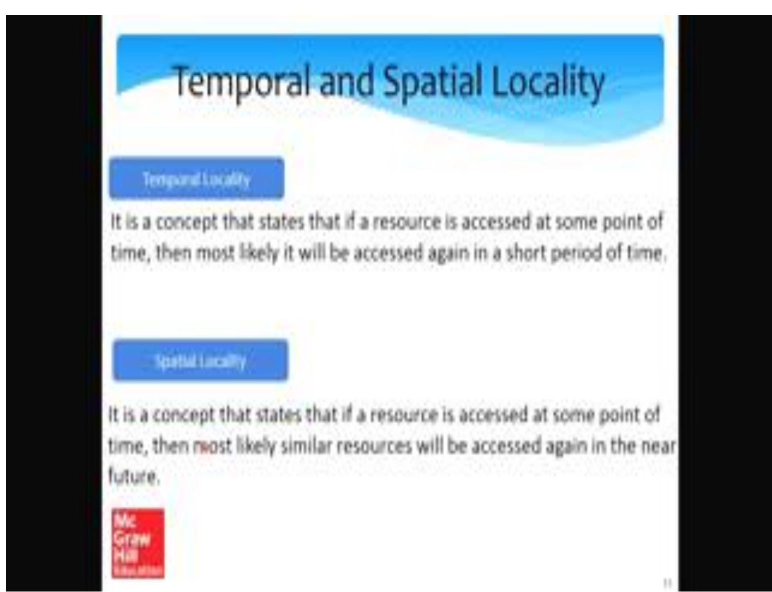
(Refer Slide Time: 24:59)

So, now let us look at some other pattern. So, let us assume that Sofia was taking computer architecture course. So, she had computer architecture books on her desk. After the course in over well the architecture books will go back to the shelf, and you know once the exam is over what do you do, you have fun you go on a vacation. So, vacation planning books will now come to the desk. So, then you know she will study where does she want to go on vacation and then figure out the right vacation destination. So, one idea that we can use over here, is that look we can bring all the vacation planning books that she has in one go. So, for example, know if she wants to let us say wants to go to Europe, but she then changes her mind and decides that she needs to go to Singapore. So, it will not be a good idea to take a separate trip for each and every vacation planning book. So, it makes sense that she brings all the vacation planning books in one go. See if she requires one of them in you know in high likelihood she will require a similar book in the near future.

So, essentially she will make a trip to the shelf, or make a trip to the cabinet and come back with a stack of similar books in one go and put them on the desk right. So, this is

what you would typically do. And even let us say when you are cooking and you know you decide that you want to cook with spices, and then it makes sense to go to shelf and bring in ginger, garlic, green chilies, everything together. Because in the more likely if you have used ginger now, you will require garlic half a minute later. So, why make a separate trip just bring all the spices in one go.

(Refer Slide Time: 27:09)



So, this is also one more pattern similar to a temporal locality. So, this pattern is called special locality. So, let us now quickly discuss both.

So, temporal locality means it is a concept that basically states that if a resource is accessed at some point of time. Then most likely it will be accessed again in a short period of time. So, we saw that example that when Sofia was preparing computer architecture exam, she was accessing the computer architecture books repeatedly over a short period of time. After the exam she decided to go on vacation. So, then she brought all the vacation planning books to her desk. So, this pattern is called special locality. So, it is a concept state that if a resource is accessed at some point of time, then most likely similar resources will access again in the near future. So, we discussed special locality in the context of a kitchen. That, let us say I am cooking and I you know decided to you know make a curry. So, then I got some ginger from the shelf, but most likely if I have got some ginger, I will get some garlic powder later.

So, I would rather bring all the spices that I have and go. So, this pattern is called spatial locality. Because there is a very high likelihood that in any cook who has used ginger will also use garlic, in a very short time.

(Refer Slide Time: 28:42)

So, let us verify that the programs that we write are similar to our behavior in real life. And they also exhibit temporal locality. So, let us define the concept of stack distance. So, let us have a stack. So, recall that a stack is last in first out structure. So, similar to you know stack of books or anything. So, let us you know quickly look at what a stack is. It is a data structure where we put in data like this, and we can just push data and also we can pop data out right. So, we have also discussed about a stack in chapter 2. It is supposing only 2 operations push and pop.

So, let us have a stack of memory addresses. So, whenever we access an address we bring it to the top of the stack. So, for example, you know address or access generational stack. So, whenever, what we do is let us say you know we are accessed one address some time ago. So, it sorts of went down the stack whenever we find it, whenever we access it once, again we search for it in the stack we remove it from it is position and we again put it at the top of the stack.

So, let us follow this algorithm. So, stack distance is defined as the number of entries between the top of the stack and where the address was found. And of course, if we do not find the address in the stack it is infinity alright. So, what is the stack distance again

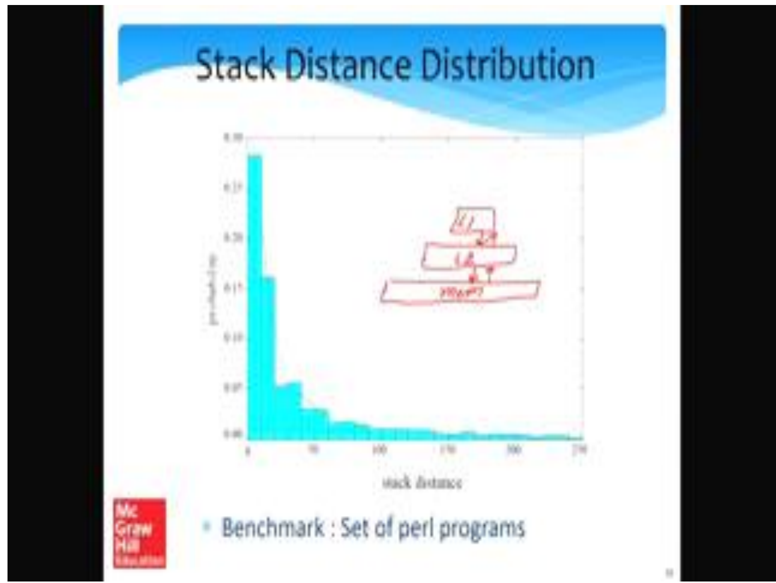
it is the distance from the top of the stack and the position in the stack where the given address was found. And if the address was not found then the stack distance is infinity. So, in the certain sense it quantifies the reuse of addresses, because if stack distance is low what it essentially means. So, if these is a stack right and let us say the stack distance is low. What this essentially means is that you know almost the similar addresses are being used again and again.

And you know this is what you would mean; if it is high means that you know addresses are being used in a very random way.

(Refer Slide Time: 31:03)

So, once again the stack distance is the distance from the top of the stack to the position at which the given memory addresses was found.

(Refer Slide Time: 31:21)



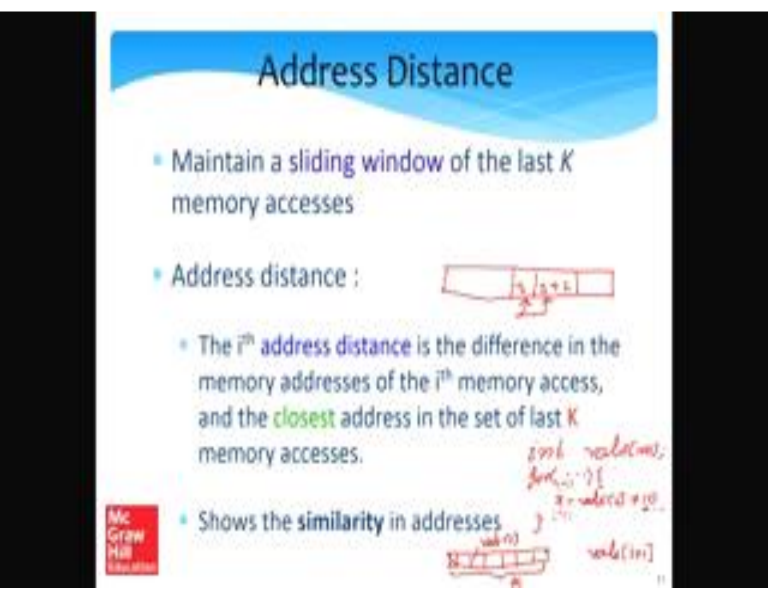
And when you find it and you take it to the top of the stack. So, well we plotted the stack distance for a given bench mark. So, almost all work clothes are similar profile. So, this was for a set of Perl programs. And for the x axis is the stack distance, and the y axis is the probability this is the probability density function essentially. So, we see that the highest probable probability of the most probable stack distances is pretty much these 2 which is makes it less than 20. So, if I have accessed one address, you know most likely within the next 20 memory accesses I will access it once again.

And that is very significant and if I just add up the rest, what this basically tells me is that you know roughly for let us see this is around 27 this is around 17, 44, 54. So, roughly for around 60 percent if the addresses within 50 memory accesses I will access it once again. So, the stack distance per se is low.

And since the stack distance is low it tells me that there is a certain amount of temporal locality in memory accesses right. And where does this temporal locality comes from it comes from the way that we quote. So, how do we typically quote the way we quote is that we write a function, the function would have some kind of a for loop, which will just run again and again, and then it will have here is similar set of addresses or the same set of addresses will be accessed over and over. Then again we will have another for loop for again the same kind of address will be accessed over and over.

So, this is exactly this pattern of the way that we write the programs, we call similar pieces of code, similar functions, access similar kinds of data, over and over in for loops and wild loops. This is what essentially gives us temporal locality. And thus we have a small stack distance. So, most stack distances are fairly low, and this indicates to us that there is a high degree of temporal locality in the set of Perl programs that we consider, but even if we were to plot this graph for other kinds of workloads, we would see similar distributions.

(Refer Slide Time: 33:57)



So, to quantify special locality let us define a term called address distance. So, let us do one thing. Let us maintain a sliding window of K memory accesses. So, which means that the last K memory accesses that a processor saw, let us just you know maintain them right, the last K accesses. So, let us define the address distance as follows.

So, the ith address distance is difference in the memory address in ith memory access and the closest address in the set of the last K of every accesses right. So, let us let us assume that K is equal to 10. I just maintain a window of the last 10 memory accesses. Now let us see I access address number 104. So, here there is no 104, but I had let us say accessed address 100. So, this means that I will chose whichever address is there in the last 10 which is the closest to 104. In this case if it is 100, I will choose this and address distance is the difference between these addresses absolute value or maybe you know just a difference and in this case the difference is 4. So, this in a sense shows the

similarity of addresses. Basically tells us that if I was in a certain memory region at a certain point of time, and then I make one more memory access how faraway is it actually right.

So, let me now explain the logic of a sliding window. Well, the logic is like this. So, how do I write programs? The way I write programs is maybe you know I would set the value of some variable. So let we consider another example with more app may be. So, let me consider an array. Let me maybe erase this and start once again. So, let we consider an array one second. So, let me consider an array here are values. So, typically the program that I write inside a for loop, we pass through the array of values and we do something on them.

So, we can may be have something as x is equal to vals i where i is the loop index you know some constant plus something else plus something else it does not matter. And then I can have some more lines after that I will come to the next iterations a for loop increment i and then access vals 5 plus 1. So, when I access vals i plus 1, I will look at my sliding window the last K accesses right. So, in the last K accesses depending on the value of K if it is chosen I will definitely find vals i in here and this will be the closest memory address to vals i plus 1.

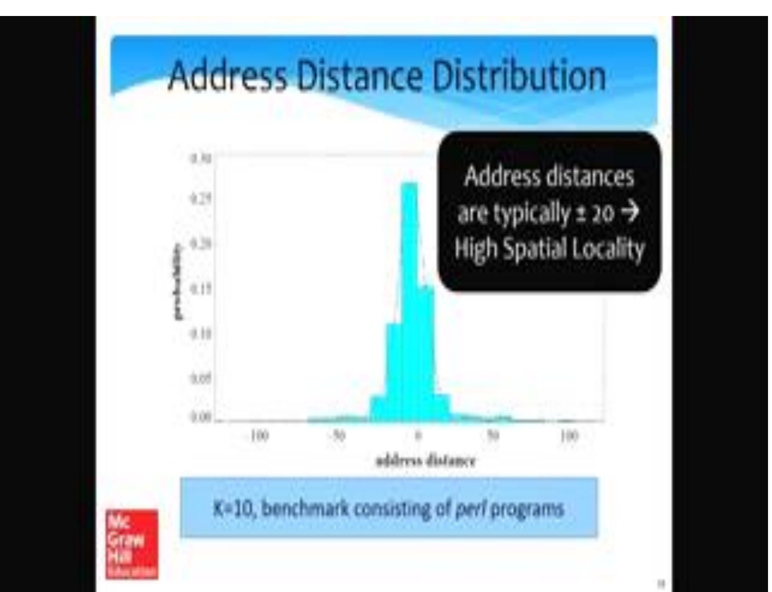
So, the difference between the addresses in the size of an integer which in most instance is 4 bytes. So, this will be the closest that address distance will be 4. So, what this is telling us is that, the first thing it is telling us is that in most programs we typically access the similar data. Similar data will be there is structures such as arrays or can be there in link list where the structures are allocated side by side in memory, but most commonly arrays and most commonly variables defined in the same region if an activation log. So, they have similar addresses. So, when I access variables with similar addresses like in this case I am accessing in one iteration vals i in the next iteration I am accessing vals i plus 1, we will see that notion of special locality is coming right.

So, in particular for the vals array since I am accessing the elements consecutively. So, if I consider the vals array, I am accessing the elements consecutively first to access i. Then I am accessing i plus 1 and so on. So, this means we are accessing similar items with similar addresses and so special locality is there, but the way we quantify it is slightly tricky.

So, I cannot compare the address the vals i with the address that I accessed last because there you know there might be several lines before it as well, which might be accessing other addresses. So, it is sort of not get confused with the way we have defined it is that let me consider a window of the last K addresses and find the closest one. So, in this case when I am accessing vals i plus 1 I find vals i which will be the closest and it will tell me that for at least vals i plus 1 the address distance is 4.

Similarly, there can be other kinds of accesses in between these lines, and we will always find that we would have accessed some other data locations which is close by in terms of memory addresses. And this is a pattern that we can use. So, this special locality is a pattern that we can use after quantifying it.

(Refer Slide Time: 40:14)



So, let me quantify, again let me consider a case where K is equal to 10 and let me consider a bench mark workload consisting on the Perl programs. And let me plot the address distance. So, if I plot, if I do a in a probability density function of the address distance. So, what I see is that the highest probability is you know centered around 0. So, that is the highest, and if I consider this region then roughly I have around 10 plus 2 5 plus 15 roughly 50 percent or more than 50 percent of the accesses are within an addresses distances of plus or minus 25 bytes.

So, this is typically the way that programs access. This is because there will be many arrays where I am accessing the array elements, there will be many local variables. So,

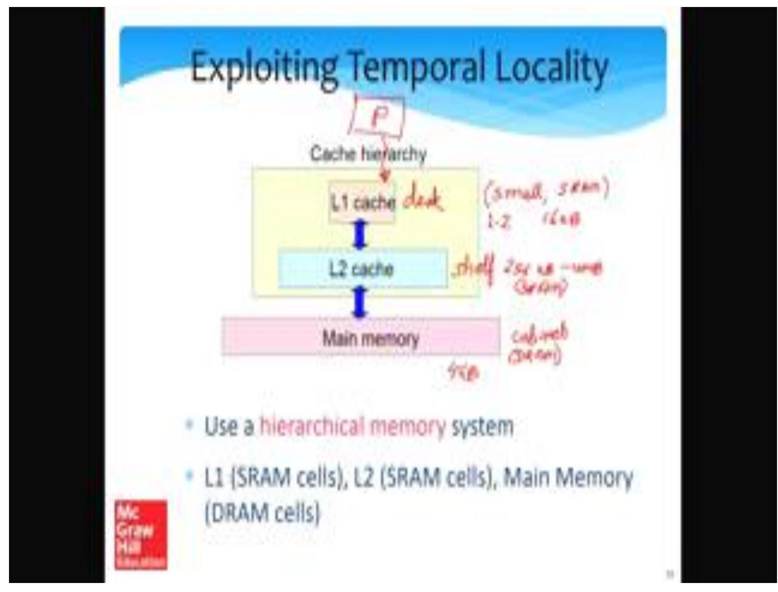
for example, if I define int you know i j and k what the what most compilers would do is that they will give them consecutive addresses, i starting at one-point j for yth later k for yth after j, accessing j and then i am accessing i and then I am accessing k. So, it does not matter what the order is, but we will always find that we are accessing similar data.

Similarly, if there is an array or there is the string or something. So, then we will also if you are doing a scan through the array using a far loop, we will also be accessing similar data. So, in notion of special locality with the low address distance is there and so that is why we get this graph, and what we see here is that most accesses, if I compute the address distance more than 50-60 percent of the accesses in within plus or minus 25 bytes right.

That is my first conclusion. And smaller is the address distance higher is the special locality. So, does this mean? This means that if I access something with memory address x very likely in the near future access something with memory address x plus or minus delta, where delta is a small number right. So, this tells me that this pattern does exists in real world programs and we should use it in some way. So, address distances are typically plus minus 20 plus minus 25.

And this tells us that if we access some data in the memory we will very likely access some other data. So, where the addresses are very close by and this is essentially high spatial locality.

(Refer Slide Time: 43:09)



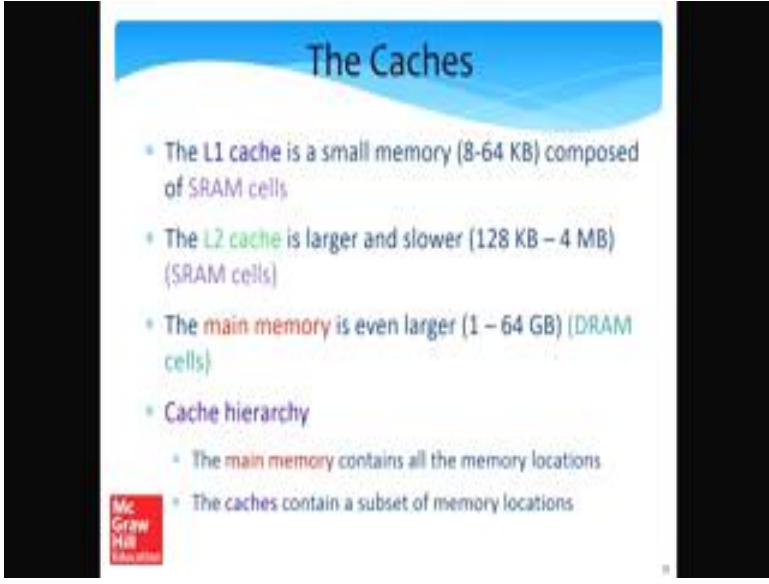
So, let us see how we can exploit temporal locality. The way we can exploit temporal locality is as follows, that instead of we can have a hierarchy of memories, where main memory is the last level. So let me may be put the processor over here to station it at the right point. So, what we can do is we can have a hierarchy of memories, say L 1 cache can be small and fast similar to the desk the L 2 cache is somewhere in the middle. So, it is similar to the shelf. The main memory is large. So, it contains almost everything that is similar to the cabinet. So, main memory is large and slow. So, similar to Sofia what the processor would do is that first it would access the L 1 cache and search for data if it finds it well and good.

Otherwise, so hopefully most of the time because of the temporal locality 90 percent of the data will be found in L 1 cache. If it does not find the data in the L 1 cache it will go to the L 2 cache which is larger and slower. If it does not find it there, then the processor will go to the main memory which is the cabinet. And how do we ensure that the L 1 is fast. So, well we make the L 1 small. So, we have a reduced capacity. So, we make it small smaller means faster. Also use SRAM cells to build it right flip flops are expensive. So, let us use small and SRAM cells.

So, L 1 will be able to access within you know 1 to 2 cycles and L 1 cache is also small. So, it is several kilobytes. So, let us say 16 kilobytes is a represented figure nowadays then the L 2 cache we can build with again SRAM cells, but it will be much larger. So, it can be somewhere within 256 kilobytes to may be 4 megabytes. And the main memory can be made with DRAM cells because we need high ram storage and we do not mind the ultra-high latency. So, this can be 4 gigabytes 8 gigabytes 16 gigabytes does not matter right. So, this is how we know we tie up our processor system with what Sofia was doing. So, the same way that she had a desk a shelf and a cabinet, we have a hierarchy of memory structures.

We have very small and very fast one typically 16 32 kilobytes called the L 1 cache right. A cache means it is a section of the memory what a cache means is that is small subset of the overall memory. So, it is small we then have an L 2 cache slightly larger 256 kilobytes to 4 megabytes, again made up of SRAMs and then we have a DRAM based structure for main memory for it is DRAM based for main memories. So, it has 4 gigabytes to 32 gigabytes of memory. So, that is a lot of memory space.

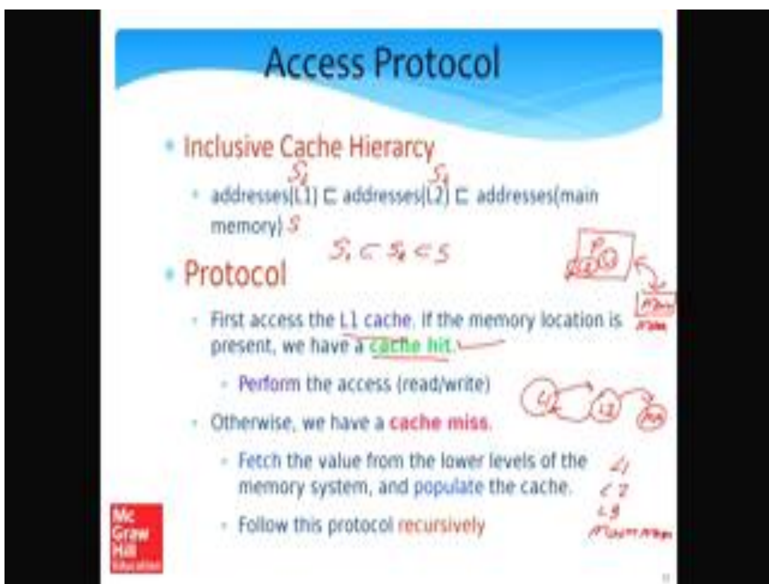
(Refer Slide Time: 46:29)



So, this is just a quick summary of what just we discussed L 1 cache small and fast L 2 cache in the middle, the main memory is large. So, this is hierarchy of caches. So, the main memory will contain values for all the memory locations, we might relax this thing later, but for the time being let us you know let us take it as the gospel truth.

That the main memory will contain all the memory locations the caches will contain subset of memory locations. So, let us assume that in a system that are 2 billion right billion with a b memory location. So, the main memory will contain all of them, but the caches will contain a smaller subset. The L 1 cache might contain a few thousand L 2 cache might contain a few million.

(Refer Slide Time: 47:21)



So, let us now discuss the access protocol and also the way that we organize our hierarchy of caches. So, typically we consider an inclusive cache hierarchy. So, this means the L 1 cache contains a subset of addresses that are there in L 2 and the L 2 cache contains the subset of addresses that are there in main memory, which contains all the addresses. So, explaining in another way the main memory contains the entire set. So, let us let this be S 1 let this be S 2 and let this be entire set S. So, S 1 is a subset of S 2 and S 2 is a subset of S.

So, this basically means that there will never be any line in any cache well any address sorry not a line address which is present in L 1 and not present in L 2 there will be never be any address which is present in L 2 and not in the main memory. So that will never be the case it will always be the case, that L 1 is the strict subset and L 2 is the strict straight subset of the addresses and the main memory.

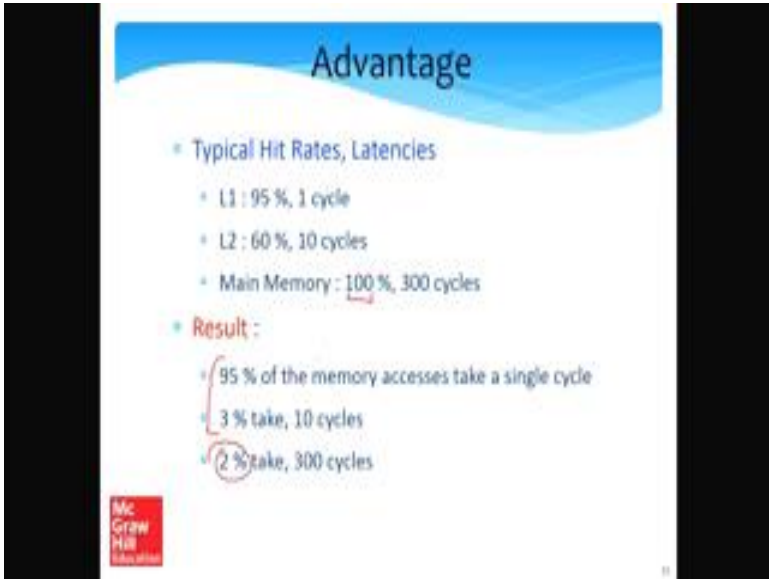
So, this could be inclusive cache hierarchy in general it is a good idea, to have an inclusive cache hierarchy as oppose to a non-inclusive cache hierarchy which causes a lot of problems. So, it is more of a research topic than actually an in a text book topic. And the protocol is as follows we first access the L 1 cache. If the memory location is present we have a cache hit, we say that the memory location is present we can present. So, we can perform the access read or write.

Now the important term here is cache hit which tells us that the memory address the memory location is there in the cache, otherwise we have a cache miss. See this is a cache miss means that the addresses are currently not there, for example, if we access the L 1 cache and the certain address is not there a request now needs to be sent to L 2. If L 2 has the address it will fetch it and give it to L 1. If L 2 does not have the address then it will send a request to main memory, right which is made of DRAM cells. And say L 1 and one 2 are typically within the processor within the chip right. So, what does the chip contain? Well the chip will contain the processor will contain the L 1 and the L 2 right. And the main memory is typically outside; you know it is a separate module which is outside.

So, if we have cache miss; so what we need to do is to fetch the value from the lower level. So, the memory system and populate the cache, and this can be followed recursively which means if l2 does not have it can send it to main memory and main memory will always have the data right. That is the assumptions that we have been making. So, we need not have 2 levels we can have L 1 then L 2 then L 3. So, lots of large processers have an L 3 as well, and they have a main memory.

Some processors also have an L 4, but L 4 is very rare L 1, L 2, L 3 is common or just L 1, L 2 and main memory is also common, but what is the most important here is to have an inclusive cache hierarchy where the addresses in L 1 are very subset of the addresses in L 2 and addresses in L 2 or strict subset of the addresses in main memory.

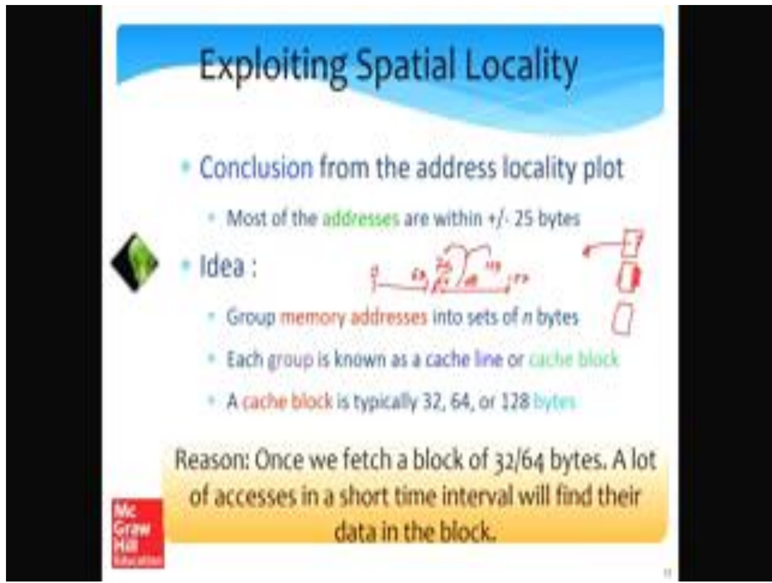
(Refer Slide Time: 51:10)



So, let us take a look at the advantage of having this kind of an organization. So, let us consider that the hit rate in L 1 is 95 percent which means that for 100 accesses enter the L 1 95 find their addresses inside L 1 and it takes one cycle. Let us then assume that the L 2 hit rate is 60 percent, and it takes 10 cycles to access the L 2. And then it goes to main memory where the hit rate is 100 percent and it takes us 300 cycles to go to main memory and get the data back. So, in this case 95 percent of the main memory accesses will take a single cycle. 3 percent will take that additional 10 cycles right to go to L 2 and get the data back.

And 2 percent will actually take that additional 300 cycles after l after accessing L 2, to get the data from main memory right. So, this is telling us that look for most of the time most of our memory accesses are fast, you know reasonably fast both L 1 and L 2. And a very small percentage of them will actually take 300. So these are very you know rough crude numbers, it is not 300 exactly it is 1 plus 10 plus 300 and, but we look at the performance of the memory system in some great detail that is the reason you know these numbers are very cruder the moment, but the important point to note is that if we hierarchy most of the accesses are will hit at the highest level, which is good. Because the highest level is the fastest, then gradually smaller and smaller percentages of accesses will trickle down to the lower level.

(Refer Slide Time: 53:12)



So, we need to design a system where the least number of accesses actually come down to the lowest possible level now this. So, the previous discussion was on exploiting temporal locality. So, how do we exploit spatial locality. So, well let us look at the address locality plot. So, the plot was over here. So, the address distances are typically within plus or minus 20 bytes plus or minus 20 or 25 bytes right. So, this means that we have fairly high spatial locality.

So, let us use this pattern. So, since most of the addresses are within a small range. So, the idea is that let us group memory addresses into sets of n bytes. Each group is known as a cache line or a cache block. So, it is an atomic unit in the sense that it will treat the entire group as one, and a cache block can typically be 32, 64, 128 bytes right a power of 2.

So, what is the reason of creating these larger blocks of bytes? The reason is as follows that once we fetch a block of 32 or 64 bytes, because of a very short address distance and because of high special locality a lot of accesses in a short time interval will fall within the block right. So, it will fall within the address range of the block or lot of accesses will find their data in the block itself. So consider this once again so, let us assume that there is this address with address 100. And most of the times we are finding in a you know short interval of time most of our accesses will be within 75 and 125. So, it is a good idea, it is a very good idea that what we do the same way the Sofia fetched all the

vacation planning books at once, because she thought she will use one, she will use the other is that we divide the memory system into 64 byte blocks.

So, one is address 0 to 63 other is 64 to 127. So, in a this basically means that if let us say we start a program of accessing program 100, in very high likelihood we will be accessing most of the addresses in the range of 75 to 125 as per our plot in a short span of time, and if you fetch the entire 64 bytes together from the lower levels from the main memory or from the L 2 cache into the L 1 cache then we will reduce a mess rate significantly right. Because the special locality if all the accesses are within this blocks in a small period of time we will all have cache hits.

So, we will not have to go outside the L 1 cache. So, it is a fantastic idea in that sense. So, what again is the conclusion from our address distance parts? It was that let us breakdown on our entire memory space that we have into contiguous sets of blocks, where a block is 32, 64 or 128 bytes and let us read a block is an atomic indivisible unit that we you know fetch from lower levels or displaced lower levels. And the advantage of bringing in a block in one go it that let us say we start with this address, then we in a high likelihood we will access the next addresses.

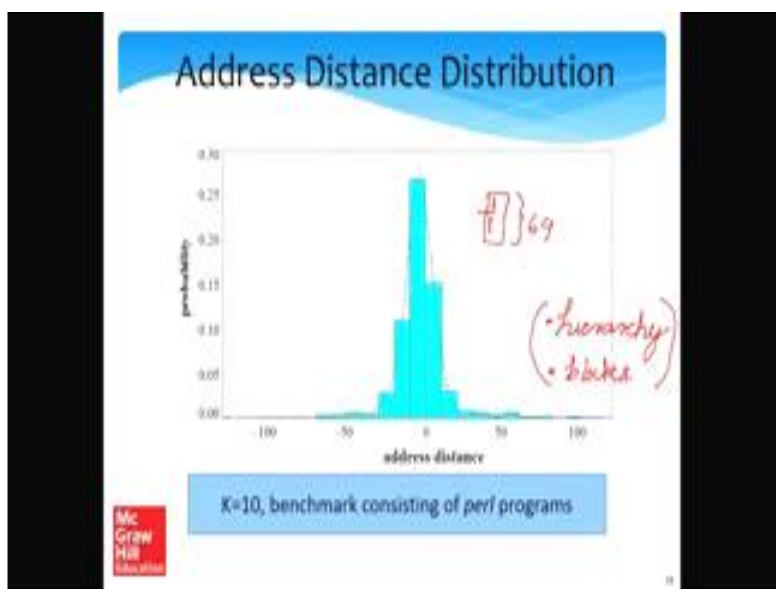
So, you have a blocks for example, contains a data of an array. So, if we start from this address very likely likelihood will scan through the arrays access all of these addresses. So, this takes care of special locality as well. So, now, we have taken a look at both temporal locality as well as spatial locality. So, next we will discuss how to use both of these a design a caches. So, let me just summarize a couple of things, and caches we will actually discuss in the next lecture.

So, what I want to say summarize is I want to go back to the plots right one for temporal locality one for spatial locality. So, for temporal locality we use the notion of the stack distance and we said that if the stack distance is low the temporal locality is high, and why was this case because use standard pattern you know we looked at kitchen we looked Sofia’s desk we looked at the program with for loops.

So, since we access similar data you know. So, over and over again in a short period of time, it makes sense to actually have a cache hierarchy where you have a small and fast L 1 which acts like a desk. So, most accesses can be served by the L 1 of you do not find it in L 1 you have a larger L 2 which acts like the shelf. And finally, you have the cabinet

which will have all the addresses. And you do not find something in L 1 you go to the main memory. So, that is how essentially by creating a hierarchy of these caches we are leveraging temporal locality, and we are bringing this desk shelf and cabinet this observation this principle into the designer processors as well. That is point number one.

(Refer Slide Time: 58:50)



Point number 2 is that we plotted the address distance distribution, which measures the similarity in addresses right that the processor is accessing found in most of them are within plus or minus 20, 25, 30 bytes. So, if we create a you know an individual unit let us say if we fetch instead of 1 byte or 4 bytes at a time we fetch let us say 64 bytes at a time, like fetching multiple vacation books at a time, if let us say we have started accessing this address most likely access the addresses nearby, and all of them I would have already fetched because they are part of the same block. So, we will have a lot of cache hits right. So, we solve the problem of you know ensuring special locality by dividing the memory space into blocks. So, that is also the very important concept.

So to summarize; after this entire one-hour lecture, we looked at a cache hierarchy right. We justified it. And we divide memory into blocks. So, these are all 2 important observations that we will take forward when we actually design caches.