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

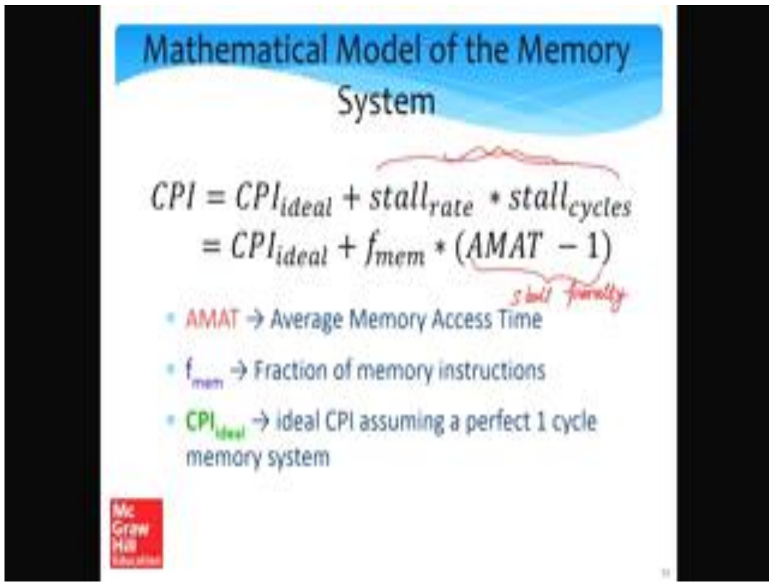
Lecture – 32
The Memory Systems Part III

Welcome back.

(Refer Slide Time: 00:26)

Let us discuss the details the memory system. Let us start with the mathematical model of what the CPI of a processor will be, if we have a memory system with caches and cache misses. Then we will discuss different kinds of misses and what can be done about them.

(Refer Slide Time: 00:45)



So, let us compute the CPI of a pipeline that has misses in the cash hierarchy. So, let CPI be the actual CPI. I let CPI ideal be the C P L with an ideal memory system, where every memory access takes one cycle and every memory access the hit. So, if this is the ideal CPI. Let the stall rate be the number of stalls that we have per instruction. So, per instruction, what is the rate of stalls that we have.

So, for example, if let say for every ten instructions, one of the instructions stalls, then the stall rate is 1 in 10, and let us stalls cycles, be the number of cycles that we waste, because of the stall. So, let us assume that, if say the ideal CPI is 1, which is the case may be a single cycle processor. The stall rate is 1 in 10, and for every stall, we on an average we used three cycles, then the CPI is 1 plus 3 by 10 which is 1.3. So, this can also be said, this is CPI ideal. So, the stall rate; say if you assume that every in instruction that goes to memory, can potentially stall. So, we can then say that the stall rate is. So, we can equate this and this f mem, is a fraction of memory instructions, and the number of stall cycles.

So, say if let say all the memory instructions hit in the l 1 cache, then the stall cycles is 0, but let say on an average, if an if all the memory instructions, the average memory access time a mat, instead of one cycles is 1.1 cycles. Then we compute the number of stall cycles as the average memory access time, minus the time it takes to perform an ideal access which is one cycle. So, in this case it is 1.1 minus 1 which is 0.1. So, as a result I

assume that if let say 30 percent of the instructions are memory instructions. So, the CPI will be CPI ideal plus 0.3 times 0.1.

So, let me discuss this equation in a different light. So, what we are essentially trying to do here in the first line is that we are trying to compute the CPI with a non ideal memory system. So, for that what we do is first we get the CPI for an ideal memory system. So, this can include pipe line stalls and everything else, but the only assumption here is, that the memory system per say has hits in one cycle. Then we use the parameter stall rate as the fraction of instructions that actually stall, and for each stall how many cycles it, is that we waste.

So, this essentially gives us a number of pipe line bubbles. So, as we have seen in the previous chapter we can compute the CPI in this manner. So, what we do now for the memory system, we take this term over here and replace it with in equivalent term, which is this. So, we consider the fraction of memory instructions in the program, and multiply them, and multiply this fraction with the average number of stall cycles per memory instruction. So, this will give us the total stall cycles per instruction. So, the average number of stall cycles per memory instruction, is the average memory access time minus 1 y minus 1. The reason that we subtract 1, is because we assume that every memory instruction, will take one cycle, and this one cycle is folded into this CPI over here.

So, we need not to double counting, and consequently we can compute the stall cycles for memory instructions as a mat, average memory access time minus 1, and if we multiply the average stall cycles per memory instruction, with the fraction of memory instructions, we will get the average stall cycles per instruction. So, then we will added to the CPI. So, essentially one thing that we see is that as the fraction of memory instructions increase the CPI will increase, the performance will go down, and similarly the memory access time increases, then also the CPI will increase.

Now why will the memory access time increase? It will increase if there are more misses in the cash hierarchy, we need to go lower in the memory hierarchy, and we need to fetch data. This will increase or over all access time, and this will increase our CPI, consequently make the processor runs slower.

(Refer Slide Time: 06:32)

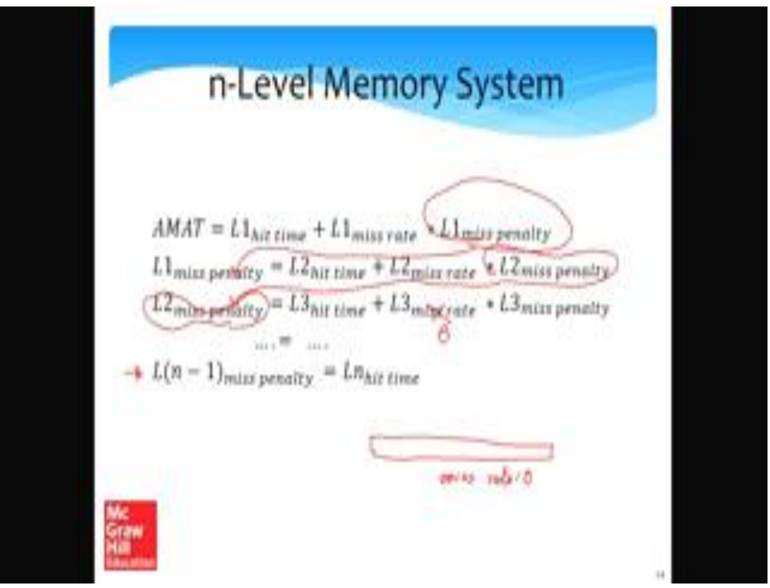
Now let us try to compute the average memory access time. So, the average memory access time can be computed as follows. So, that is, take the hit time. So, every instruction needs to go via the l 1 cache. So, let us consider the time the hit time of the l 1. So, this will be added for every access irrespective of the fact, whether it is a hit or a miss. So, fraction of the accesses that go to the l 1 cache, will actually miss in the l 1 cache, and for all of the all of those misses we need to pay a miss penalty, which is the number of cycle say takes to go down the memory hierarchy, and fetch a value and write it back to l 1. So, the l 1 miss penalty can further be broken down as follows. So, the l 1 miss penalty corresponds to this quantity.

So, let us take a look at it. So, the average memory access time is again the l 1 hit time plus the l 1 miss rate multiplied with. So, in the l 1 miss penalty all the accesses then miss in the l 1 cache go to the l 2 caches. So, we have a similar formula for the l 2 cache. For every l 2 access we have l 2 hit time, because irrespective of a hit or miss, we will take some time to figure out whether it is a hit or miss. So, this is the l 2 hit time, plus we will have. You know some of these accesses will again miss in l 2. So, l 2 miss rate, and similar to an l 1 miss penalty will have an l 2 miss penalty, which is the number of cycles it takes to go down the memory hierarchy and fetch a value. So, this is pretty much the formula for the average memory access time, and of course, we can go down. So, will have that in the next few slides, with the important thing to note here, that irrespective of a hit or a miss we need to spend some time, this is the hit time.

And then you know as fraction of the accesses will miss, and for the misses we need to pay miss penalty. So, when you are actually computing the stall penalty, which is the quantity in the previous equation, which is pretty much this, the stall penalty. So, say if you consider, I am sorry for this quantity, it is this quantity, this, it is this quantity. So, this quantity will be the average memory access time minus the l 1 hit time. So, the reason that we define it like this is because we assume that in an ideal cache, sorry in an ideal processor, all the instructions will hit in the l 1. So, the l 1 hit time will you folded into the computation of CPI ideal. So, we need not consider it again, in the sense we need not consider it as a penalty.

So, as the result it should be ignored. So, it's not a stall penalty right. So, it should be ignored when we are computing the stall penalty, the stall penalty will essentially with the average memory access time minus the l 1 hit time, which means that this quantity over here is this stall penalty, and this quantity can further be expanded as l 1 miss rate times l 2 hit time plus l 2 miss rate times l 2 miss penalty.

(Refer Slide Time: 10:22)



So, what we can do is that we can further extend this equation. So, as we have shown in the last slide the average memory access time, is the l 1 hit time plus the l 1 miss rate times the l 1 miss penalty. The l 1 miss penalty if further is the l 2 hit time plus the l 2 miss rate times the l 2 miss penalty. So, in a every single term is getting defined like this, and furthermore the l 2 miss penalty is after the l 2 if we have an l 3 cache, is the l 3 hit

time plus l 3 miss rate times l 3 miss penalty. Say if we have n levels in the memory hierarchy right, n levels of caches.

So, for the last level; so for the second last level, the l n minus 1 miss penalty will be the l n hit time, because for assuming that the last level of the memory hierarchy contains all the lines, and as a result it is miss rate is 0. So, this is the main memory that we have been talking about, it is miss rate is 0. So, basically this quantity, you know this miss rate quantity over here become 0. So, as a as a result for the last level, the miss penalty of the second last level is the hit time of the last level, because we are assuming that no accesses miss over there.

So, this further tells us, that if you know the miss rates at each level, and also the hit times at each level, we will be able to compute the average memory access time.

(Refer Slide Time: 12:00)

Let me now define some terms. So, the miss rate that we have been using is called the local miss rate. So, it is equal to the number of misses in a cache at level I, divided by the total number of accesses at level i. So, let me give an example let us assume, that we have an l 1 cache and we have an l 2 cache. Let us assume 100 accesses go to the l 1 cache and then 90 of them have a hit in the l 1, the 10 go to is l 2. And from the l 2 to the main memory 5 accesses go. So, at the l 2, the local miss rate is essentially is it gets 10. So, it has 5 misses, and it gets ten accesses. So, the local miss rate is 50 percent.

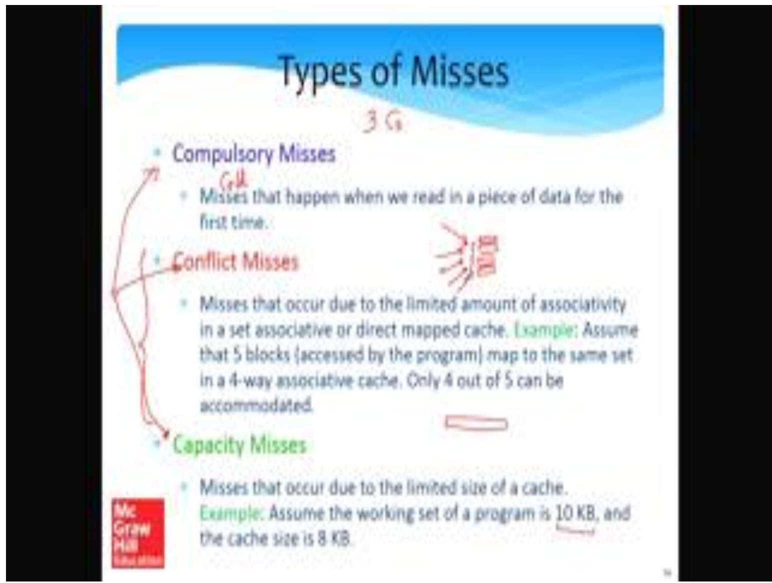
In comparison with can define the global miss rate, it is equal to the number of misses in a cache at level I divided by the total number of memory accesses. So, the global miss rate will essentially be, for the l 2 cache 5 divided by 100. So, what is typically of more importance is the local miss rate, which is 5 divided by 10 or 50 percent, and that is what we have been using in all our formulae, but it is still important to you know this term global miss rate, which is the misses at a certain level divided by the total number of memory accesses. Now let us define the term could working set. So, before I define that I want to talk about how typically a program works.

So, the way that a program works, is we typically have you know multiple for loops and while loops. So, we have lot of iterations, and then we move to another section the program. Again we have a lot of for loop iterations, you moved to another section and so on and so forth. So, may be in this part, we will access one array or a group of arrays. So, for a certain in a fix period of time will be accessing some data. Again will move to another part of the program will be accessing some other data. So, in a given time interval the amount of memory that a program accesses. This amount of memories called a working set is called the working set of the program at that particular point in time.

So, for example, let us assume that given program, accesses typically; 1 megabytes, 2 megabytes of data and it does not access more than that. We can say that the working set is 2 megabytes. Let us assume that one more program accesses 2 megabytes for some period of time, then another period of time it accesses 4 megabytes, you know 4. It accesses 4 megabytes meaning that it has loops that sort of scan through that entire 4 megabytes of data, and then again it accesses to like that. So, we can say this point of time the working set is 2 megabyte, at this point the working set is 4 megabytes, at this point the working set is 2 megabytes and so on.

So, essentially you know this again an approximate definition is nothing formal about it, but roughly in a small time interval in the execution error program, the amount of memory that it accessed, is called as the working set. So, what we want is that we want the working set, to fit in the caches such that most of the data that a program what require, would be available in the caches.

(Refer Slide Time: 15:57)



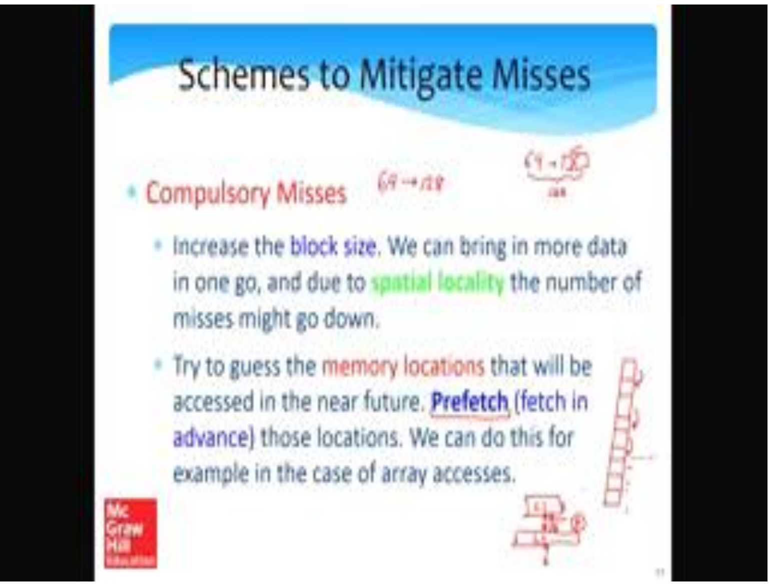
Now, giving the fact that we have seen this, let us look at misses and what can be done to them, done with them. And let us try to classify the types of misses that happen in a memory system. So, these are 3 cs actually, because in a three times of misses, all three start with the letter c. So, these are called a 3 cs. So, the first categories of misses are compulsory misses. So, these are misses that happen, when you read in the piece of data for the first time. So, there also called cold misses. So, compulsory misses are cold misses will happen, when for examples in the program is starting up. Non other data will be in the caches. So, we will need to read it for the first times in this, these misses have to happen. So, they are called compulsory misses, also called cold misses, because essentially when a program starts it cold.

Next we can have conflict misses; say conflict misses like this that these are misses, that occurred due to limited amount of associatively in a cache. So, for example, assume that a cache is 4 ways associative, which means that in each set there are 4 ways or 4 entries. The access of the program is such that may be, you know you very frequently access five separate addresses, and 5 separate cache lines, that are mapping to this set. So, since this set can only contain 4 entries and there are 5 accesses to 5 separate blocks for caches line same thing. One of them will need to get displace the other one will come in. So, we will have a lot of these evections, because only in a 4 out of those 5 can be accommodated. So, we will have a lot of evections, and what is happening, there is a conflict, there is an address conflict inside a set. So, this is known as a conflict miss.

So, the last category of misses or capacity misses. So, assume that, you know the working set of a program is 10 kilo bytes, and the size of the cache is 8 kilo bytes. So, in this case the entire working set will not fit. So, you know assume that it is a 10 kilo byte log array, and we are doing a lot of operations on that array, but a cache is only 8 kilo bytes. So, the entire array will not fit, and as a result we will have a lot of cache misses. So, this is called a capacity miss.

So, what are the three kinds of misses again, we have compulsory misses which happen when we are reading in program for the first time and we will have a miss. We will have conflict misses which are basically, because are cache ways or limited associatively, and hence in a will have some conflicts, because we can a store a lot of blocks in one cache. We limited by the associatively and as a result if something new comes in, something old has to go out. And we will have capacity misses where if the working set exceeds the size of the cache, then we will have evictions replacements, and consequently higher cache basis.

(Refer Slide Time: 19:31)



So, let us look at schemes to mitigate misses or to reduce the number of misses. So, compulsory misses, so what is the idea; one way of doing reducing this is, we can increase the block size. So, let say instead of a 64 byte block size, I make it a 128 byte block size. then this means are in one go we are searching in more data from the lower level, and because of spatial locality, this is the reason that we made blocks in the first

place, it is possible then the number of misses might go down, because you will access a lot more you will find within the same block, and this would not have been possible, had the block sizes been smaller.

So, let me give an example. So, let us assume the block size is 64. So, you fetch in 64 bytes, and after that we need to access the next 64. This is not there in the cache. So, will have a cache miss, but assume that we fetch both the 64 consecutive 64 byte blocks together, and your block size is 128, then essentially all the 128 bytes are available in the block. as a result we will not have you know the cache miss that we had for the second block. So, this can work. The other is let us try to guess, let us have a small circuit that tries to guess the memory locations that will be accessed in the near future. So, you know may be if you are accessing in arrays, let us consider it typical array access. So, you may, considering. Sorry for drawing a slanted array. So, let us assume that we have one program in (Refer Time: 21:19) bubbles sort.

So, then it will, we are accessing the array elements sequentially one after the other. So, if you have a small circuit over here, which is sort of analyzing the array accesses, analyzing the addresses of the array. It will be able to predict, that if you have reached this point, then most likely, and since we are accessing the mean sequence. So, most likely in, we will access the next few locations. So, it can guess that the next you locations will be accessed, because we are going consecutively, you are going in increasing order of array locations. So, it can guess and fetch them from the lower level and advance, and this is called pre fetching. pre fetching means guess predict what will access in the future, and fetch the data from the lower levels of the memory system, which means that if we have an l 1 cache, and we have an l 2 cache, and we have a small pre fetching circuitry over here.

Say it can look at all the accesses that are going into the l 1 cache, and on the basis of that make a guess that what kind of accesses are required. So, then you know put in request of its own to the l 2 cache, and that data will be again this applied to the l 1 cache. So, hopefully this can reduce the case, the number of misses in the l 1.

(Refer Slide Time: 23:07)

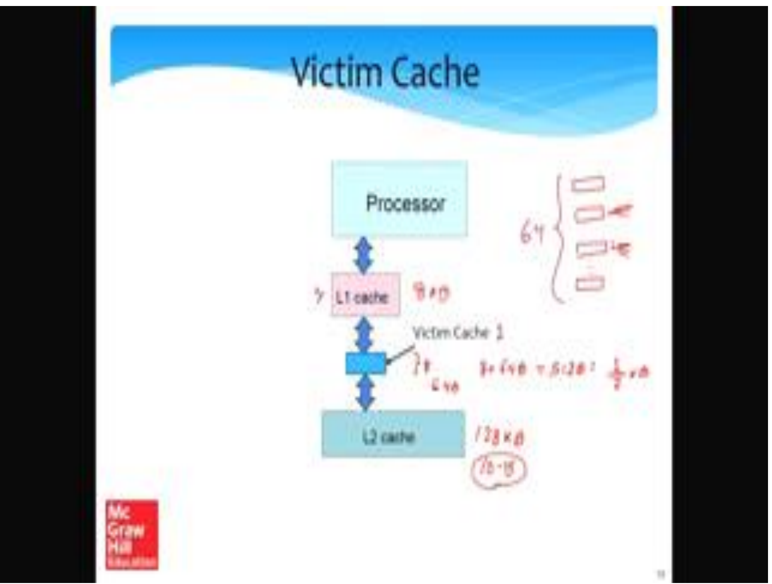
Now, let us talk of us scheme to reduce the number of conflict misses. Say in this case this simplest solution; the simplest solution is to increases the associatively of the cache say if it is, because the problem is essentially coming because of associatively. If we have a limited 4 way associated of cache and five lines are mapping to the same set, we will have misses. So, let us increase it from 4 ways to 8 ways, which means have 8 entries in each set.

Well this is easier set then done, because more or the number of ways more is the power consumption, because of more comparisons and more is the latency. Again because of the fact that we have more circuits, you know there is a certain slow down that is increased. What we can do instead is that we can use a smaller fully associative cache called the victim cache. So, the idea of a victim cache is like this, in any line that gets displaced on the main cache, can be put in the victim cache. So, what we can have is we can have an l 1, we can have an l 2; of course, they are communicating, but we can have very small cache over here called a victim cache, and we can have many heuristics to you know put data here, but essentially anything that is being thrown out of the l 1 cache, can be put in the victim cache in a, it is a victim. So, it can be put over here, and subsequently any read access needs to check in both the l 1 and the victim.

And once on the victim fills up, we can evict something here and we can make it b sent to l 2. So, there are certain advantages of the victim cache I will discuss this in the

subsequence slides. let us take a look, before discussing that let me take a look at the third point, which is to write programs in a cache friendly way. What this basically means, is that we can, you know either at the level of the programmer or at the level of the compiler we. So, this is meanly at the level of the compiler actually. So, say write programs, it is not that much of programs point of view, but it is from the point of the view of the compiler, that allocate your data structures in such a way that they are fairly balanced across all the ways of the cache; so no all the sets of the cache. So, no set is unduly stress right there is no amount of undue traffic, on each set, and this will reduce. So, you know this is sort of reduce the pressure on anyone set and decrease the number of misses.

(Refer Slide Time: 25:48)



So, let us discuss the victim cache in, little bit a more detail. So, let us consider the first case; say we send a request and we find the data in the l 1 cache right. So, basically in this case nothing needs to be done. We read the data in l 1 cache and the data goes back to the processor. Now let us assume that we do not find the data in the l 1. So, why we will be not find the data in l 1, because it is not there, but let us assume that previously some line was thrown out from l 1, instead of writing to the l 2, we wrote it to the small fully associative cache, called the victim cache, which is small and fast.

Say in this case if first check in l 1 and we check in l 1 and victim parallely does not matter, and if we find it in the victim cache, we take it and we have given to the

processor. So, once again we find the data in the victim cache, we take it and we give it to the processor. If we do not find the data in the l 1 and the victim, well then we need to go to the l 2 cache, and give it to the processor. So, so what is the advantage of a victim cache. So, the advantage is like this, so there is. So, let us at look you know may be some sizes the l 1 is the 8 kilo bytes to l 2 is 128 kilo bytes. The victim cache might just contain 8 entries, and a 1 entry is 64 bytes. So, 8 entries will be. I wanted 12 bytes or essentially half of kilo byte.

So, half a kilo byte is not really a lot, it is not in a tremendously increasing the size, but it can make a fair amount a difference, and the reason that I say so is like this. So, let us assume that you know as we have been seeing in a running example, we had 128 entries. So, if we have two way associative cache. So, will have a 64 sets. So, if we have 64 sets. So, let me may be draw them. So, you consider, let us consider a two way set associate of cache with 64 sets and each set has two lines right, each set of two cache blocks; so the ideal case with these when the entire access is balance. So, each of the set gets roughly a similar number of cache accesses; however, if that is not the case, then what should be done is the. well then what will happen is that maybe there will be some such sets which are pretty unlucky, it may be, could be this one and this one.

Fair it has you know, fairly large number of accesses. So, it is proportionately large number of accesses go to these two. So, there will be a fair amount of conflicts and these two sets and if some conflict misses. So, if some of those lines can be accommodated in the victim cache. So, the victim cache mind you has half a k b of size in this case. Then we can sort of in a logically think, that first some of the sets that of high contention we have in a sense increased their size.

Or in other words what it means is that for some of the frequently accessed lines that we still half to through away from our cache, because of limited associatively. We will find another home called the victim cache, which is again a very small and very fast structure, and this will hopefully have an effect which is far higher then, you know what it is size would otherwise predict.

For example half a k b cache when especially sandwich between in 8 k b and a 128 k b cache will have a minimal effect, but if we design the algorithm to sent cache lines to a victim cache, so slightly judiciously. and mainly target lines there are evicted from

highly accessed or high contented sets, then most likely it is lines will be accessed more frequently in the future, because of in a temporal and special locality effects. And as a result the victim cache might actually play significant role in appreciably reducing the number of accesses that go to the l 2 cache. The memory request at go to the l 2 cache.

So, as the result a victim cache can proved to be very useful, because you know it is not significantly adding to the access time with l 1 takes, cache takes one cycle to access, and we accessed l 1 and the victim parallely, or we accessed the l 1 first in the victim later. We will still, you know that the access time to victim cache will still be around in a additional two cycles, but now as compare to that in l 2 cache can take much longer to access somewhere between 10 to 15 cycles. So, that is still the significant amount to speed up, and in the victim cache can capture those lines which are; otherwise frequently used, but because of conflict reasons, all of them cannot be accommodated in the same set. We will definitely see some benefit.

Thus the reason the victim cache idea is very popular, and it is used you know it is a very common artifact in modern processors.

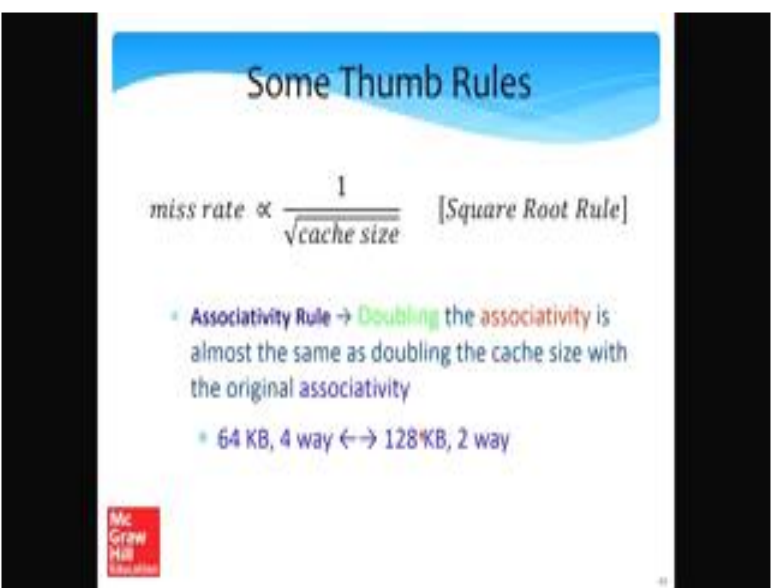
(Refer Slide Time: 31:39)

Now, let us discuss some other schemes to mitigate misses. So, let us look at capacity misses. So, we have already discussed compulsory. So, what was the scheme for mitigating compulsory misses? It was either increase the block size or prefetch, the scheme to mitigate conflict misses was in an increase associatively; that is easier set that

done, because the higher associatively means in a more latency, more time, and more power. So, in comparison a victim cache could be used, and the victim cache as we just, you know discussed if we can design an algorithm.

So, there are many algorithms in the literature. So, there are many in research papers which can. So, which essentially ensure, that some of the most frequently accessed lines which cannot be accommodated in l 1 cache, find a place in the victim cache. We can see significant improvements in the performance of the memory system all right. So, for capacity misses we again have a simple algorithm, the simplest thing is increase the size of the cache; that is again easiest set then done, the reason being that a larger cache is also slower and more inefficient in terms of power, or we can use better pre fetching techniques. So, in this case what can happen is, if we can predict and fetch what is needed in the future. So, hopefully that will sort of reduce the impact of having a smaller cache.

(Refer Slide Time: 33:25)



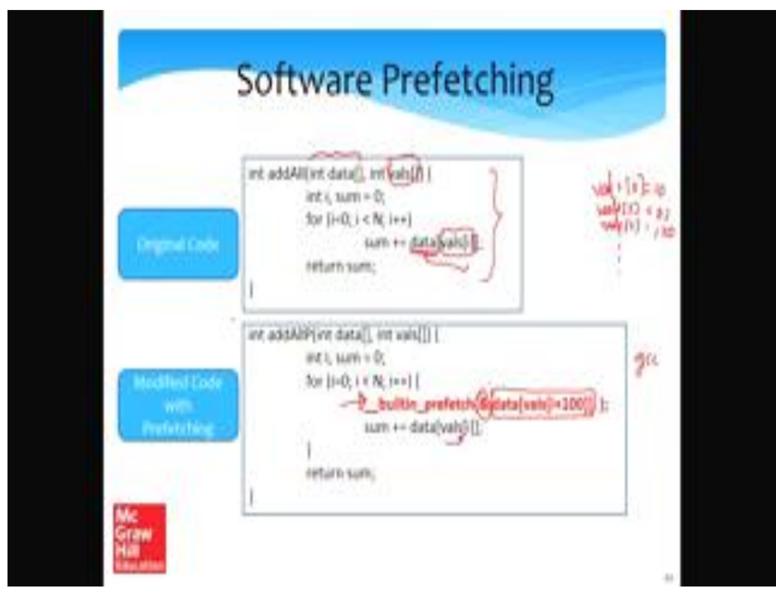
Prefetching is person not a part of this course; it is a part of an advanced course and computer architecture. So, let us now discuss some thumb rules. So, what is the thumb rule? The thumb rule is something that people have observed, where the thumb rule is something that people use, but it does not have a theoretical proof. It is just in a something that of come from observation, and that is the reason is called a rule of thumb, it is not a proof; I mean it does not have a proof. So, one thumb rule is that the miss rate

is proportional to, the inverse of the square root of the cache size right, which means that if I. this is called a square root rule. So, which means that if I double the cache size or if I increase the cache size by 4 times, then I will reduce the miss rate by a factor of 1 by 2. So, I will reduce it by 50 percent. So, the miss rate is 10 percent, and I want to make the miss rate 5 percent, I need to increase the cache size by 4 times.

So, in any other set an approximate rule, so might not be 4 times, might be three and half, might be five, five and half. So it is a good you know it is a good strategy to at least remember. The other is the associatively rule. So, this cells. So, this is again also approximate no theoretical proofs, just has been observed. You know observed in simulations in actual processors, and it is also you know very approximate in nature. So, the associativity rules is, the doubling the associativity, is almost the same as doubling the cache size, having the original associatively. to given example is 64 k b four way cache, will roughly have the same miss rate, very similar miss rate, as the 128 k b two way cache right.

So, what are we doing? We are doubling the cache size, and we are reducing the associatively way half right. And so, in this case, or if we go from this way to this way, doubling the associatively, from two way to 4 way is the same as, in a doubling, keep the original associatively, where double the cache size. So, well these two rules are approximate can be used, you know whenever required.

(Refer Slide Time: 36:05)



Let us now move to the next slide, where we will look at, what is called software prefetching. So, there are two kinds of prefetching algorithm; one is software prefetching and the other is hardware prefetching. So, in hardware prefetching what happens is, that we have a hardware circuit that looks at the memory accesses, and then tries to guess what will access in future, and it tries to fetch that from the lower levels of the memory system. We can instead do it in software and it is very easy. So, we can actually argument us c code, to fetch things from the memory system in advance, well before we actually use them.

So, let us take a look at this program over here. So, in this particular function what we try to do, is that we consider an array. So, we consider a data array, and we consider in array with the indices, are call the array walls. So, we iterate through each and every entry of the wall arrays. So, we assume that there are very n entries, where n is a pre defined global constant. For each of the n entries starting from 0 till n minus 1, we access each entry of the vals array, and in each entry of this vals array what we do, is that. So, we assume that each of entry of the vals array contains in the index. We take that index and we accessed the data array, and whatever number is stored over there, we added to the sum, and finally we written the sum. So, this is called add all function.

So, we analyze this function, we will realize that this function neither has temporal locality, nor does it have special locality. So, why does not it have temporal locality. Well so, this will again depend on what is contained within the vals array, but it will contains fairly random elements. So, the same elements or similar elements, similar blocks are not being access frequently over time, in the same period of time. So, as a result temporal locality not there; special locality is also not there, because in a consecutive elements are, elements with nearby addresses, are also not being accessed. Well the vals array is being accessed. So, there is special locality in terms of accessing the vals array, but not in terms of accessing the data array. So, the vals array we are accessing vals 0, then vals 1 and so on.

Where assume val 0 contains 10, vals 1 is 32, vals 2 is 120, then the addresses that we are accessing are 10 32 and 120 which are absolutely random. So, for the vals array we have special locality, because they are increasing (Refer Time: 39:21) very indices, but not for the data array, and for the data array we pretty much have very random addresses. So, as a result even special locality and temporal locality, both are not there for the data

array, even though they are there for the evolves array to some extent, but we will. And because entries are being access randomly in the data array, we will have a lot of cache misses.

So, this means that. So, such kind of code, you know fair we have a double in direction, in the sense the index of the data array is it iself in array element. This is common, this is very common in scientific code is very common, and it is call in irregular array access and this typically has a very bad cache performance. So, what we can do, is that we can add one line, which is make a cache performance fantastic, and it will. So, some experiments indicate that performance of the program can even increase by three times. What we do is that, if you are using the GCC compiler on Linux, this thing only holds for GCC on Linux, can hold for GCC on Mac also, but I have not tested. We can use the function underscore underscore built in underscore pre fetch.

So, this is a software prefetch instruction, in the sense that, given in address it tries to pre fetch the data, and tries to bring it to the highest levels of the memory system which is the l 1 cache, and if the address just in case is wrong or is invalid does not matter. So, there are never any problems so the address as invalid right. So, that does not create any problems at all. So, what we can do is that we can prefecth some address, which will be accessed with the future. For example, we can prefetch this address over here, which is vals i plus 100. Assume that I am you know in a current iteration we are at vals i. So, after hundred more iterations, will be accessing the data array entry at vals i plus 100. So, we can prefetch that entry in the current iteration, by calling the function underscore underscore built in prefetch.

So, what this does. So, the way that this works is that, we send a pointer to it. So, the operator in c essentially creates a pointer, and I mean it gets the address, the memory address. So, we get the memory address of this element, and we send it to the memory system; such that, and from the lower levels of the memory system. We can the data. We can get the data back up to the upper levels. So, this instruction of course, we are adding an extra instructions. So, the performance penalty is there, but hopefully by pre fetching data in advance, it will become very easy; not very easy, but by pre fetching data in advance. So, when we actually accessed the data later, we will find it at the upper levels of the memory hierarchy, and the miss penalty, or the memory accessed time will be much lower.

So, this is actually the case, say if you just run this program, you know I invite all the readers, all the listeners to write this program and test it for themselves. So, will also find the some of these programs on the website at the book, and definitely inside the book also. So, if these programs are written and they are tested you will find that there is a 2 100 2 3 and 8 percent improvement in performance, mainly because we are pre fetching data, and particularly this pattern is hard to pre fetch in hardware, because the hardware has to guess, that all the in indices that are going in to the data array, or actually coming from another array called vals, and that is pretty difficult to for the hardware to guess, even though it is possible there are research works in that respect, but doing this in software is still easier, it is slightly more intuitive, and so what we do is, that for any iteration we research data that will require hundred iterations later.

So, readers can ask a question that when we are towards the end of you know, when your close to end, then that value plus hundred will be invalid value; that is correct, but even always pass in invalid address to built in pre fetch, it will not cause any problems, it will not cause the program to shut down, or have a falter you know or the segmentation for it is. So, these issues will not happen. So, we are pretty much safe.

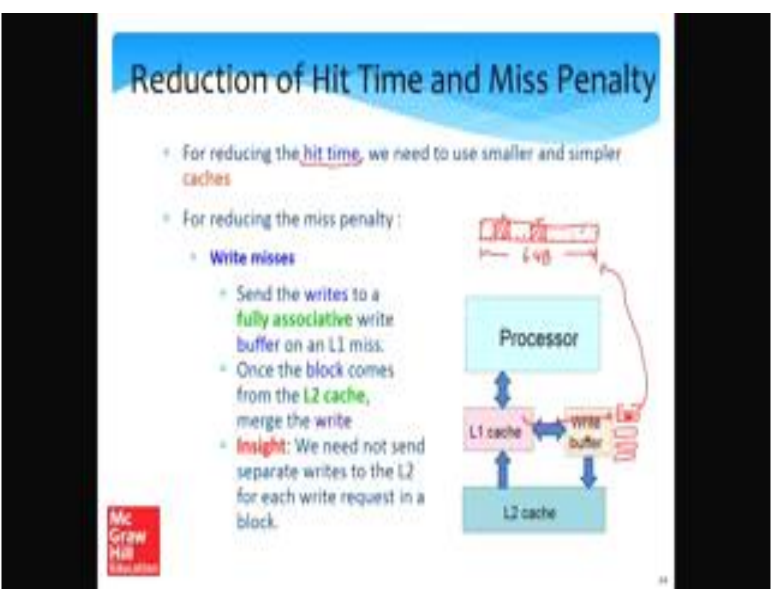
(Refer Slide Time: 44:17)

Knowledge considers hardware pre fetching. So, in hardware pre fetching we have a pre fetcher that takes a look at. We look in take a look at two pieces of information. So, it

can either take a look at, let say the accesses to the l 1 cache. So, that is one thing that it can look at, or it can look at the misses that are happening in the l 1 cache.

So, basically it can be any one of them, but looking at the misses slightly more common. On the basis of that, the pre fetcher can decide, which are the addresses so the processor will most likely access next, it can send then the message to the l 2 cache. So, get in those lines to the l 1 cache; such that the processor will find them ready and available. So, the hardware pre fetcher, it works in some cases where the access stream is predictable, if that is not the case then we need to use software pre fetching.

(Refer Slide Time: 45:18)



There are other things that we can do, to make the processor faster by essentially reducing the memory access, average memory access time. So, we can reduce the hit time as well. So, the reduce the hit time, we need to use a smaller and simpler cache, but well having a smaller and simpler cache, has other negative effects as well, and the main. The biggest negative point here is, of using a smaller and simpler cache, that it reduces the hit rate as well.

So, this trade of needs to kept in mind. Additionally we can try to reduce the miss penalty. So, the miss penalty can be reduced as follows that let us consider write misses. So, if you consider write misses, where you know we might write to successive words in the same line. What we do is that we create a small write buffer, with a very small

structure 4 or 8 entry just after the l 1 cache. So, we send. So, write buffer in not be after the l 1 it can be between the processer and l 1 as the well.

So, it does not matter the concept is the same. So, we send the writes to a fully associative write buffer, on an l 1 miss. So, basically this essentially you can 10ds multiple blocks. So, then we do a write. So, basically what we do is that if you are writing to 4 bytes. So, we write some part of it, by the rest of the block we do not know, because the data has not been fetched.

Once the data is fetched from the l 2 cache, we populate the rest of the contents of the block. So, the insight here is, that we need to send, and also in the, at that same time if let say another request comes. So, may be in a let me just expand this sort of put a magnifying glass. So, let us consider each block to be a 6ty 4 bite (Refer Time: 47:22). So, let us assume that the processor wants to write to some 4 bytes in the middle, after that the professor once to write another 4 bytes over here.

So, you know extreme of writes are coming. So, the a naive approach would be that we first get the data from the l 2 cache, and then we pretty much write it to the write buffer. We get the data from the l 2 cache to the l 1 cache in then we do the writes, but this one take time. So, what we can instead do is, that the writes can go to an entry in the write buffers. So, let say we can write to, you know we can allocated the entry and write 4 bytes.

Similarly we, if another write comes in the same, for the same block we can write the other 4 bytes, and we can wait for the line to come from the l 2 cache. Once it comes we would have known that this chunk and this chunk I have already been written to, but the rest of the contents of the block can be filled with whatever comes from the l 2 cache right, which is essentially this entire dotted region. So, the advantage here is, that we do not have to wait, and we can do or writes and then we can proceed with the next instructions right. So, in a certain sense it is released us the miss penalty of the l 1 cache, especially when it comes to writes.

(Refer Slide Time: 48:52)

Let us now look at reducing the read miss penalty; these two optimizations critical word first in early restart and use together. So, the insight is something like this, that the memory word that causes the read write miss. So, basically let me go one step back. So, let us consider the size of a cache block. So, the size of a cache block is typically 6ty 4 bytes, but when we read an integer that might only be 4 bytes. So, basically in these 6ty 4 bytes, if let say we wanted to read a integer and we did not find it in the l 1 cache, we would only be interested in 4 bytes right, even though the entire block needs to be fetched, you know as an optimum measure in indivisible unit, but we are primarily interested in 4 bytes. So, what we can do is that, we can slightly tweak the system here.

So, consider the l 1 cache and consider the l 2 cache. Say if the l 2 is supplying 6ty 4 bytes to the l 1. So, how will it supply? Essentially there is some copper wires between l 2 and l 1, and it can take multiple cycles. So, you know typically, we can only transmit something like 4 bits at a time. I am sorry 16 bytes at a time that is. So, it will take around 4 cycles to transmit. So, what we can ensure is that the memory word, 4 bytes of the memory word. So, the memory word that we are interested in, is transmitted first right. So, maybe we can divide the 64 bite chunk, in 2 in a 16 bite smaller chunks, and may be the memory word that we have is present in this 16 bite chunk. So, we can transmit this, the first.

Once we transmitted we can go for in the early restart, which basically means that the l 1 cache, can fetch the for 4 bytes within this 16 bite chunk, and send it to the processor such that the processor gets it is data, and it can restart it execution. So, the processor does not have to stall. The advantage here is, that we are able to get a little bit of mileage or leverage, little bit of mileage out of the process. And the way we are doing it is that we are teaching the order in which the bytes get transferred from the lower levels of the memory hierarchy to the l 1 cache, and instead of reading the bytes sequentially within the block, we read that part that we are interested in, and transmit it the first. After it reaches the l 1, we immediately give it to the processor; such that the processor can restart it is execution.

(Refer Slide Time: 51:49)

Technique	Application	Disadvantages
large block size	compulsory misses	reduces the number of blocks in the cache
prefetching	compulsory misses, capacity misses	extra complexity and the risk of displacing useful data from the cache
large cache size	capacity misses	high latency, high power, more area
increased associativity	conflict misses	high latency, high power
victim cache	conflict misses	extra complexity
compiler based techniques	all types of misses	not very generic
small and simple cache	hit time	high miss rate
write buffer	miss penalty	extra complexity
critical word first	miss penalty	extra complexity and state
early restart	miss penalty	extra complexity

So, this is the simple slide that summarizes all are techniques. So, this is like a one stop place for all are techniques. So, let me just go over it. So, for compulsory misses we realize that the best technique is a large block size, because if we have a larger block size you are fetching more data at once, and because of special locality some of it might be useful. Pre fetching both software as well as hardware is a generic technique, very useful, can be used to reduce; it is actually all kinds of misses with primarily capacity and compulsory misses. So, it is involved with a little bit of extra complexity, and say it reduces the risk of displacing useful data, but nevertheless pre fetching done wisely has it is benefits.

We can increase the cache size, this will reduce capacity misses, but larger is the cache higher is this latency, higher is this power, and higher is this area, so that is negative points. To reduce conflict misses we can always the increase the associatively. So, no doubt, it will reduce conflict misses, but will make the cache more complicated. We need to compare with more tags. So, this is the associated with higher latency and higher power. The victim cache is almost always a good idea, and the reason is almost always a good idea, is because is very smalls, you know it does add the little bit of extra complexity. So, the overhead is minimal, and but it is effective does can vary depending on the type of the program, but it primarily reduces conflict misses, especially when some sets get a lot of traffic. In that case if we can say some ways in the victim cache will get some benefit.

We can also use compiler based techniques to reorder the code; such that cache misses get reduced. So, this is anyway in advance topic is out of this scope of this book, it is not very generic, but it is extreme the effecter. So, after talking about the 3 c's, three kinds of misses, let us talk about hit time. So, you want to reduce the hit time, you have to go for is a smaller and simple cache, but again this increases the miss rate. So, as the trade of here, let us consider the miss penalty. So, for writes we can reduce it with write buffer. So, here what we do is, that we write to the write buffer and we just continue our execution. When the word comes from the lower level, the new data is merged with the older data. So, sort of helps us go ahead, and while reading we need to check both the write buffer as well as the original cache.

And for read misses, we have a combo of these techniques early restart and critical word first. So, that is some amount of extra state, some amount of extra complexity, but the advantage is, that it allows us allows the processor to go head, because we give the preference to that data that originally caused a miss, the 4 bytes or 8 bytes with the processor wanted to read, and it cause the miss, if you give preference to that data it helps us restart the execution the programs sooner.

So now, that we have seen the details of the memory system will go to the last part; the fourth part of this lecture series which is on virtual memory.