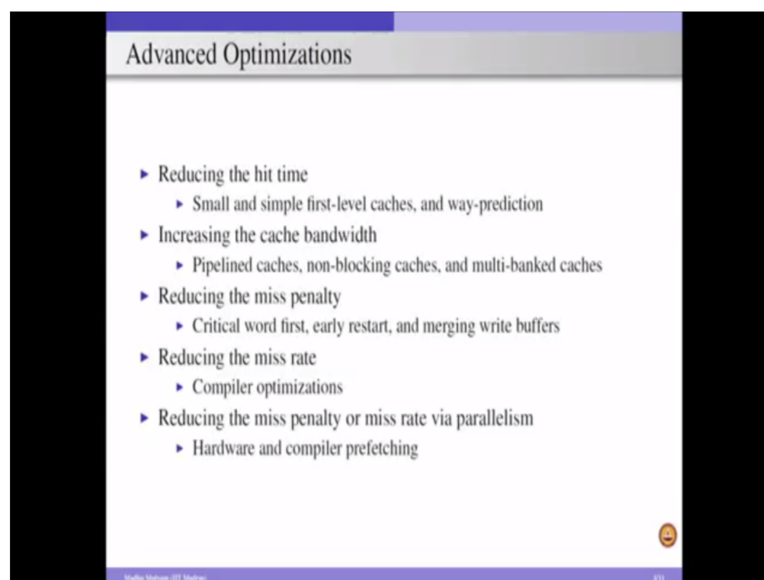
Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras.

Module – 03
Lecture – 09
Memory Hierarchy Design (Part 4)

So, in the last module we discussed basic cache optimizations and now we are going to locate some advanced optimizations to the cache memory. So, previously we considered the techniques to reduce the hit time, reduce the miss rate and reducing miss penalty, but now we also add the bandwidth improvement techniques and the power minimization techniques to this list.

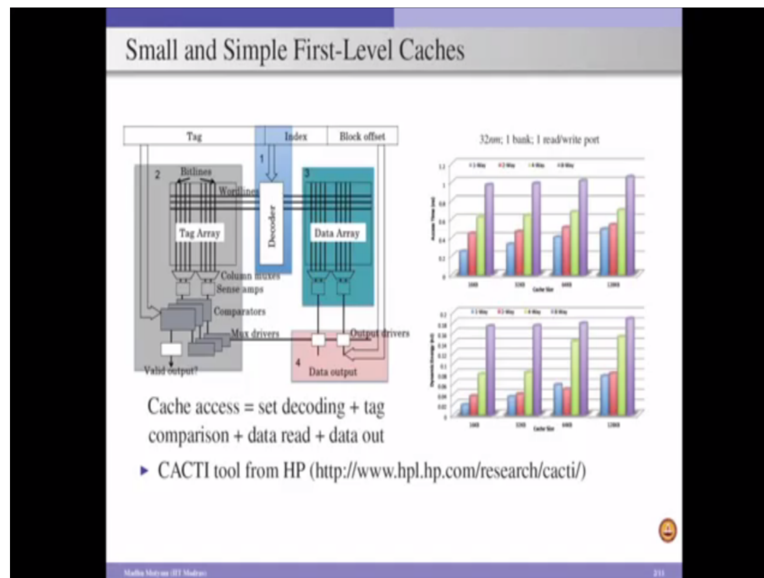
(Refer Slide Time: 00:42)



So, first we start with reducing the hit time. So, to reduce the hit time we will consider small and simpler caches. We also consider a technique called as way prediction and for increasing the cache bandwidth we consider pipelined caches, non-blocking caches and multi-banked caches, to reduce the miss penalty we look at techniques such as critical word first, early restart and merging right buffers. And for reducing the miss rate, we look at compiler optimization, optimization from the software not from the hardware itself.

And also we look at the prefetching techniques, both the software and the hardware prefetching techniques for reducing the miss penalty or the miss rate. So we start with hit time reduction techniques.

(Refer Slide Time: 01:40)



So, we know that a cache consists of a tag array, data array, a decoder and set of other components for performing a read or write operation on it and it supplies the data to the processor. So, from the time the processor issues address to the cache, to the time at which the cache supplies the data. So, there are several components involved in it. And each of these components are going to consume some time and the overall time is going to determine your hit time.

So, effectively cache access consists of going through the set decoding, tag comparison, data read and data out. When I say tag comparison, it consists of tag array read and then comparing with the tag in the address of the memory request. So, effectively if the size of the cache is large, automatically our word line length will be increased and bit line length will be increased and similarly, the decoder width can increase and because of all these things out overall access time is going to increase.

To reduce the cache access time we have to reduce each of these things. But before that we look at what is the actual access time for a given size of the cache with a given associativity. To do that we actually consider a tool from HP is called a CACTI and which actually provides the access times for the cache memory as well as the DRAM based memory. And it also provides area as well as the energy consumption and these energy values will be given for each of the components in the cache. So, first we start with the access time and here this graph shows different sizes of the caches starting from 16 kb to 128 kb.

And we considered one way that is a direct mapped cache, a 2 way, 4 way and 8 way associative caches. And we also assumed that the entire cache is a single bank. And so we consider one read write port per bank and we model the entire cache using 32 nanometer process technology. Remember for different process technologies, the access times and the energy values are different. So, for our calculations we consider 32 nanometer process technology.

From the graph, we can clearly see that for a fixed size cache, for example - consider a 16 kb cache, if I increase the associativity the access time is increasing. And similarly, for a fixed associativity, for example - consider a 2 way associative cache and as I increase the size of the cache access time is increasing. That says that having a simple cache, with the size of the cache is small then, we are going to get a better hit time. But remember if I reduce the size of the cache or if I reduce the associativity of the cache, the capacity misses or the conflict misses can increase.

So, effectively when you want to freeze in a particular associativity and the size, you have to look into all these things also into consideration. So, as I mentioned earlier that the entire computer architecture course is effectively a design space exploration. You have to explore all these things and then finally come up with a better design point. So, because these decisions have to be taken before, much before, actually fabricating the chip.

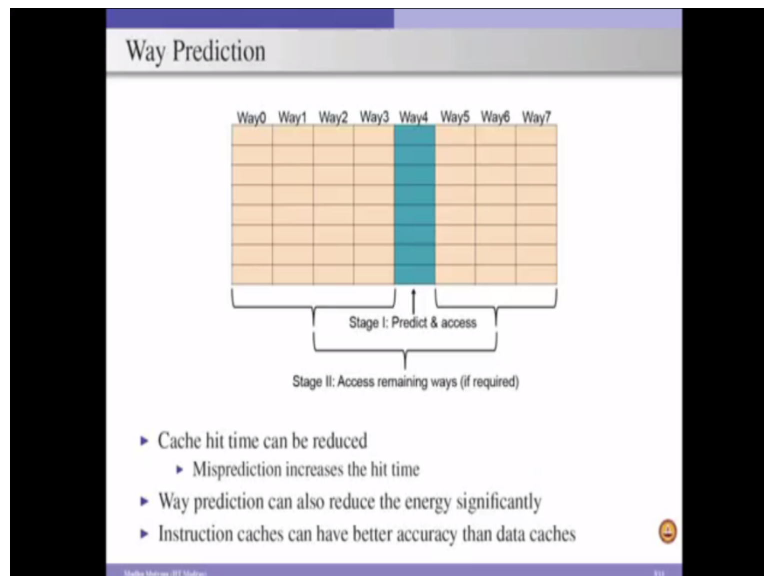
So, for these things, for modeling these things, we typically use simulators. And in this particular case we consider a CACTI tool. So, this shows that access time is reduced if I consider a small size cache with a smaller associativity. Now, we look at the energy point of view. Again we consider only the dynamic energy, whenever we perform any access, this access is going to consume some energy and that energy is called the dynamic energy. If I increase the size of the cache for a fixed associativity the energy is increasing.

The reason is as the size of the cache increases because our, the bit lines as well as the decoder length is going to increase and that is actually contributing to the increased dynamic energy. And similarly, if I consider a fixed size cache, but I increase the associativity, then also the energy is increasing. This is mainly because our, as the associativity increases our, word line length is going to increase and that is actually contributing to more energy. And also another point is because here we are considering, accessing both the tag and the data simultaneously.

As the associativity increases our data access energy contributing to the overall dynamic energy is increased. So, this also shows that to reduce the overall energy of the cache, we have to go for smaller and simpler caches. By the way, the details about the CACTI tool and how to download and the technical report about the CACTI tool and how it works and so on are available in this particular website.

And it is an open source tool anyone can download and play with the cache simulator for different configurations and you can see so how each component in the overall cache contributing to the overall energy and the overall access times and so on.

(Refer Slide Time: 08:30)



So, the second technique to reduce the hit time is a prediction mechanism. Consider an 8 way associative cache. We know that if at all the data is present in the cache, it will be present in a particular set, that too only one way of that particular set. So, once we know that the data can be there at most in one way why are we accessing all the ways. If we are going to access only one way, we can perform these in a direct mapped mechanism. So, accessing one way is going to take less amount of time compared to accessing all the ways, but to do this or prediction mechanism should be very accurate.

As long as our way prediction mechanism is accurate, we are going to reduce the overall hit time, but what happens if our prediction is wrong. When the prediction is wrong then we have to search all the other ways of that particular selected set, to see whether processor request can be serviced by this cache or not. If even those other ways also incurs a miss then

we have to go to the next level cache or the memory to supply the data, but now the overall performance of this way prediction technique depends on the prediction accuracy.

Higher the accuracy we can improve the hit rate significantly and the overall performance can be improved. If the accuracy is very bad then automatically for most of the requests we are going to incur multiple accesses. First is access for the direct mapped cache type of access and the second one is, for the all the remaining ways. Because this is going to increase the overall time, because the first one is we are going to access a selected way, followed by all the other ways as long as our prediction is giving a negative result.

So, when we want to use this way prediction mechanism we have to come up with high accuracy way prediction mechanisms. And whenever there is a miss prediction as I mentioned earlier the overall hit time is going to increase significantly. And also as long as the way prediction is going to give you high accurate results, then we do not have to access the remaining ways in the data array portion of the cache that reduces our energy consumption also.

And typically this way prediction technique produces high accuracy in the case of instruction caches. As instruction request exploit more spatial locality, but in the case of data caches requests may exhibit not so high spatial locality. So, the way prediction techniques may not be effective.

(Refer Slide Time: 11:20)

Pipelined Caches

- Pipeline cache access to improve bandwidth
  - Pentium: 1 cycle; Pentium Pro – Pentium III: 2 cycles; Pentium IV – Core i7: 4 cycles
- Makes it easier to increase associativity
- Increases branch misprediction penalty

Now, we look at pipelined caches to improve the overall bandwidth. So, we know that a cache access goes through several components. The first one is we will go to the set decoder, then the tag array access then comparing the tags with the tag in the address and then access the data array. And then read the data and finally, apply the block offset to get the requested word. And effectively all these components are independent to each other. So, we can simply apply our pipelining concept.

By the way, we are going to discuss these pipelining concepts in the unit 3 that is fundamentals of pipelining. And we will see how pipelining improves the overall throughput of the bandwidth. Of course, we consider the instruction pipeline there, but the pipelining concept can be applied for any component. So, here in this particular case we apply the pipelining concept to the caches.

So, we divide this entire cache into multiple pipeline stages. And 2 subsequent stages are separated with a pipeline register. So, whatever the data we read from one particular stage will be stored in the intermediate pipeline register and that will be given to the next stage and it continues. So, here in this particular case we consider a 4 stage pipeline, but if you consider different Intel processors Pentium was using 1 stage pipeline.

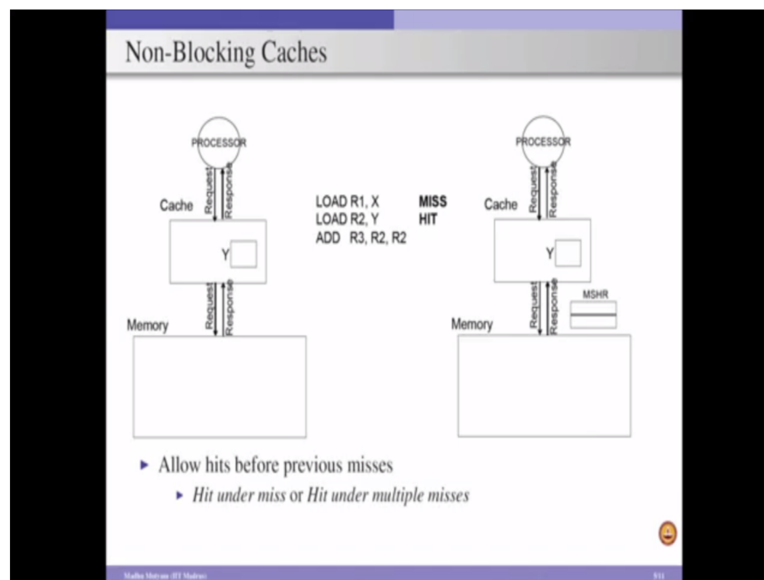
So, entire cache was considered as a non-pipeline, Pentium pro to Pentium 3 were using 2 cycle pipeline cache because as the capacity of the cache increases, we know that the access time is going to increase. And if we keep this entire thing as a single pipeline cache then it is having an impact on the processor frequency. To reduce the impact on the processor frequency one option is we have to go for a pipelining of this cache. And that is what they have considered in Pentium pro to Pentium 3. And in Pentium 4 they were considering 4 cycle pipeline cache and that is still continued even core I7 processors which are latest processors from Intel, which also uses 4 stage pipeline cache.

Of course, it improves the bandwidth and also it gives an option for us to go for increased associativity in the cache. Especially, when we are dealing with the parallel mode of access in the cache, when we have a higher associativity, we discussed earlier with the increased associativity, the access energy is going to increase significantly, but when we consider a pipeline cache because our tag array is accessed before the data array access. So, as a result based on the hit specified by the tag array access, we are going to access only the required way in the data array portion of the cache.

So that the energy consumption will not be significant and that gives us an option to increase the associativity. So, there are advantages with the pipeline caches, but it also gives some disadvantages. Especially in the case of branch miss predictions and again we are going to discuss these branch predictions, branch miss predictions when we come to the unit corresponding to exploiting instruction level parallelization. Because once we predict a branch that is going to take place and we fetch the instructions and pump these instructions into the instruction pipeline, but after some time if we realize that the predicted branch is incorrect then automatically we have to flush the entire pipeline. But as we increase the pipeline stages for this cache that also increases the overall pipeline stages of the entire processor.

So, as a result our penalty due to miss predictions in the branches is going to increase significantly in our pipelined caches. So, that is the reason why when we are considering these pipeline caches, we have to be careful enough to consider how many stages we have to consider so that branch miss prediction penalty is not significant.

(Refer Slide Time: 16:10)



So, now we will consider non-blocking caches. So, first consider a blocking cache, where when a processor issues a load or a store request, request goes to the L1 cache and if the L1 cache cannot supply the data for a load request from the processor, it sends the request to the memory and while the request is serviced from the memory, cache will not take any more

requests from the processor. If the cache is working in this mode then this is called as blocking cache. Cache is blocked for previous request to be serviced.

Until the previous request is serviced the cache cannot take anymore request from the processor. And we know that servicing a request from a memory is going to take significant amount of time. So, as a result the blocking mode of cache degrades the processor performance significantly. And all the current processors are typically consider a non-blocking mode of operation. So, what is a non-blocking cache? So, consider example where there are 2 load request, generated by the processor.

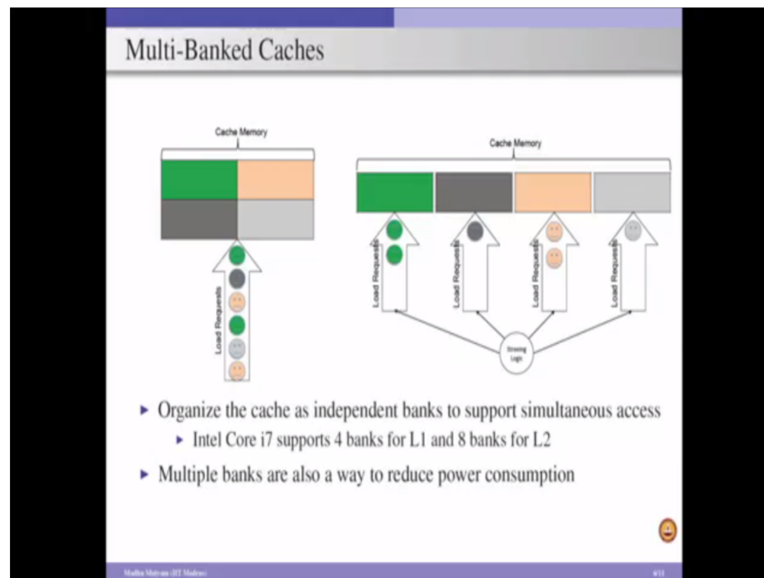
And then there is an add operation on the data what we read from the previous load request. And assume a scenario where the first load request is a miss in the cache, but the second load request is a hit. So, in the case of blocking caches, even when the second request is supposed to be hit in the cache, we cannot process this request. But when I consider a non-blocking cache, while the previous request is a miss in the cache, the cache supplies this miss request to the memory.

And while memory is supplying the data for this miss request from the cache and the cache accepts the next request that is Load R2, Y and 'Y' is a hit in the cache. So, the second load request is a hit in the cache. So, the cache supplies the data for the second load request to the processor. While it is transferring data to the processor, memory supplies the data for the previous load request. To do these non-blocking operations, all we have to do is, we have to maintain some set of buffers, which are called as miss status handling registers.

Whenever there is a miss in the cache, we report an entry in this MSHR and send that request to the memory. So, all the requests which are miss in the cache, will be reported in the MSHR. And an entry is made in MSHR and based on the availability of the memory, MSHR will read one entry at a time in the order and it supplies this data request to the memory and memory will process that particular request.

And whenever the memory supplies the data again this data is searched with the corresponding address in the MSHR and matching entry will be deleted from the MSHR and the data is supplied to the L1 cache. So, as a result we can improve the performance of the overall system. So, it allows hits before previous misses it is also called as hit under miss or hit under multiple misses.

(Refer Slide Time: 19:50)



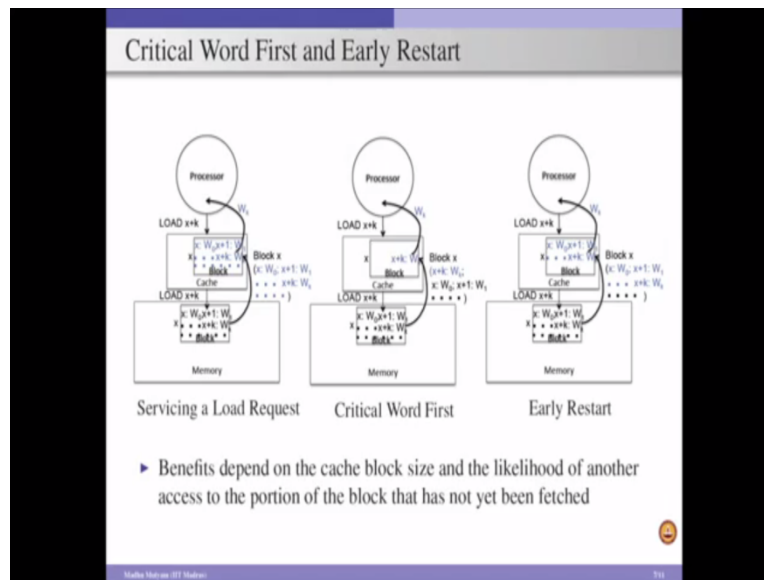
And we can improve the bandwidth by designing cache in a multi-banked width. For example, consider a cache which is designed as a single bank or a monolithic cache. And there are several request, load request from the processor. And we have considered this in this particular example, the cache is colour coded and the request also colour coded. So, that the colour of a request which is matching with a portion of the cache indicates that this request is satisfied by that particular portion of the cache.

So, when you have a single monolithic cache. So, we have to service these requests one after another. Here the assumption is our cache can take at any point of time only one request. Of course, we can service multiple requests by using a single monolithic cache, but we need to have a support of multi-ports. If we have multiple read ports for the cache, then the cache can service multiple requests simultaneously. As long as these requests are not colliding with the same address or as long as these requests are not conflicting with each other, but a multi-ported cache actually increases the overall energy as well as the access time.

So, may be you can consider this CACTI tool and consider multiple ports and see what is the access time versus considering a single port. You can understand the statement what I said previously that multi-ported caches takes more access time as well as increases the access energy. So, the other option is without increasing the energy as well as without increasing the access time, we can improve the bandwidth by dividing this cache into multiple bands like this.

So, we divided the cache into multiple banks. So, that the request generated by a processor will be steered to particular bank based on the address match. Of course, it is going to add a steering logic time when a request comes from the processor we have to see which bank the request can go. And then we will send the request to the corresponding bank. So, that, that particular bank can service the request. Once we have 4 banks, this cache can service 4 requests simultaneously. And each bank can have one port. So, ports we are not increasing. So, as a result it is not increasing the access energy and access time. And since multiple banks can service the request parallely so, we can improve the overall bandwidth. So, if you consider Intel core i7 processor, it supports 4 banks for L1 cache, 8 banks for L2 cache and as I said earlier, it reduces the power consumption associated with this multi banking.

(Refer Slide Time: 22:57)



So, now consider another method, the critical word first. So, in a normal the cache when there is a miss in the L1 cache, we go to memory and memory is supplying the entire block of data to L1. So, after the entire block is loaded into L1 then L1 cache supplies the requested word to the processor. And consider a scenario where our block size is 64 bytes. So, until the data is available from the memory to the L1, L1 cannot supply the data to the processor, but actually processor is waiting for only 64 bits of data.

And it is not waiting for the entire 64 bytes of data to be transferred from memory to the cache. So, as a result what we can do is, when the request is sent to memory from the L1 cache, we identify the location in the particular address location. We identify the

corresponding block in the memory and we will go to that block and we take the requested word and transfer this word to the L1 cache and which in turn supplies this requested word to the processor.

While it is transferring the remaining words will be transferred one after another. Remember, memory cannot supply all 64 bytes of data in single cycle to the L1 cache. It happens bit by bit, it typically transfers couple of words in one cycle and then reads the next couple of words of data and so on. So, as a result there is a serialization happens when we are reading the data from memory block in the memory. So, it is going to take significant amount of time. And to minimize this penalty, the miss penalty, we can supply the requested word that is the critical word, which is required by the processor as early as possible. So, that processor can resume operations of a following request while the memory is supplying the remaining things.

So, we are effectively overlapping memory transfer time with the processor computation time so, that the overall CPU time can be reduced. This is one way of doing that the other one is, we can go for early restart. Without reordering our the byte transfers we continue transferring data from memory to the cache, byte by byte in the sequence, but as and when we come to the critical word, that is required by the processor or the word which is addressed by the processor. We supply that to the L1 cache and L1 cache immediately transfers that data to the processor.

So that, we can overlap the remaining byte transfers from the memory to the L1 cache with the processor computation. So, the early restart is simple to implement, but it may not overlap the operation significantly. Whereas, the critical word first overlap memory transfer time with the CPU computation time, but it is a bit complex. You have to reorder the things and you have to locate the critical word and then transfer that and place that in an appropriate location in the cache block.

And then rearrange the remaining words coming from the memory into the cache. So, the operations are bit complex in the critical word first. But anyway so the benefits of this critical word first are early restart especially depends on the block size as well as the chance of having further requests into the remaining portion of the block, which is not yet transferred from the memory.

If the block size is, let us say, a one word, you do not critical word first for early restart. Of course, a one word block we are not we are not going to consider because of the performance

reasons and so on because to exploit the spatial locality we consider multi-word blocks. Because as the block size increases our transfer time increases. So, if we send the critical word first, we can improve the performance. And again, for example, if the processor is requesting the last word of a cache block, which incurred a miss in the L1 cache.

So, we send the request to the memory and we are supplying the critical word first, that is the last word is supplied to the cache first and which in turn supplies the data to the processor. So, processor is going to resume with the remaining operations, but the remaining operation is another load request which is to the first word of that particular block and which is not yet transferred. So, then what is going to happen is, the processor is again incurring another miss and so on. So, effectively this all depends on the overall performance of these 2 techniques depends on the block size as well as what is the probability that, the processor generates a request to the non-transferred words of this particular block. So, another way to reduce the miss penalty is considering the merging write buffers. We already discussed the write buffers.

(Refer Slide Time: 28:35)

Merging Write Buffers

Write Address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write Address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

- When storing to a block that is already pending in the write buffer, update the write buffer
- Do not apply to I/O addresses

Typically the size of the write buffer is very limited may be it consists of 4 entries or 8 entries. And each entry can store 4, 64 bits data. Now, consider a scenario where we have a 4 entry write buffer and each entry is storing 8 words of data and each word is 64 bits here. Now, processor generated write request and these are to different addresses, one is address location 100, the next one is address location 108, 116, 124. And assume that our cache is designed as a write through cache. So, whenever we are performing these write operations.

So, we are writing to the L1 cache as well as we are writing to the write buffer between the L1 and the L2 cache.

So, as and when the processor issues a request for address location 100, we write to the corresponding location in the cache and also we dedicate one entry for this address location 100. After that a processor generated a next write request which is to the next address 108. It will be written to the cache in the appropriate location and also next entry is allocated in the write buffer and the data is written to that particular thing.

Similarly, there are 2 more requests, after 4 requests no more space is there in the write buffer. So, that if there is any further write request form the processor, we cannot continue further, unless we make some room in the write buffer. To make a room in the write buffer, now we have to take an entry from the write buffer and write it to the lower level caches and free that entry. So, that processor can resume with subsequent write requests. So, this is the normal way of performing operations on the write buffers, but if we provide some intelligence to our processor such a way that, whenever there is a next request comes from the processor.

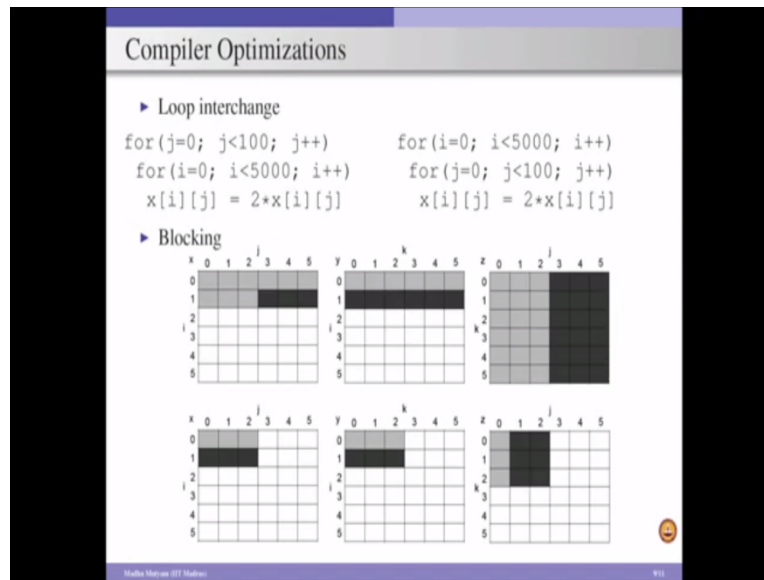
We just compare that address with the modified entry in the write buffer, to see whether this address can be merged with the previous addressed data. So, that is called as merging write buffers. If you see this example, the processor is writing to the address location 100, address location108, 116 and 124. So, if we merge all these things and still we can say like, processor is writing to address location 100 but to different components in that and all these are contiguous.

So, we just give one entry in the write buffer. So, that the remaining 3 entries in the write buffer are free and whenever L2 cache is free we take this entry and we write all these things together to the L2 cache. Also writing large chunk of data to the lower level caches is much efficient compared to writing one word at a time. So, that way also we can reduce our overall miss penalty. So, when storing to a block that is already pending in the write buffer update the write buffer.

So, this is the overall idea of merging write buffer concept. But this technique cannot be applied for I/O addresses because the addresses of I/O registers are not contiguous in memory. So, as a result we cannot apply for the IO addresses and this write merging is typically applied for only the addresses which are contiguous in the memory as like the

example shown in this file address 100, 108, 116 and 124. These are contiguous and we can apply our write merging peacefully for this. So, in order to improve the overall performance we can also take the help from the software. So, that is we can exploit compiler optimizations in improving the performance of the cache memory. A simple example is loop interchanges.

(Refer Slide Time: 33:03)



Consider a case - we have an operation performed on 2 dimensional array, where we have a nested loop.

$$
\begin{aligned}
&For(j=0;\, j<100;\, j+=1)\\
&For(i=0;\, i<5000;\, i+=1)\\
&\quad X[i][j]=2*X[i][j]
\end{aligned}
$$

The data is stored in the memory in such a way that it follows this order x[0][0], x[0][1], x[2][2] and. So, on x[0][99] then x[1][0], x[1][1], x[1][2] and so on x[1][99] and it continues. When we store the data in that order in the memory, but if we apply this particular code what is going to happen is, we access x[0][0] first, then we are going to access x[1][0].

Note that x[0][0] and x[1][0], these 2 are separated by 100 words gap. And these 100 words cannot fit into a single cache block. So, as a result we incur a miss for the second request and then we have to bring in that block into the cache and then supply the data to the processor to perform this operation. So, as a result the spatial locality cannot be exploited efficiently, if we consider this particular piece of code. To increase the spatial locality we can just interchange these loops.

So, that the for(i=0; i<5000; i++) will come first, followed by for (j = 0; j < 100; j++). So, as a result we now request the data which is like x[0][0], x[0][1], x[0][2] and so on. So, when we bring in a block of data to service first data item, we also bring in subsequent multiple the elements of that array. So, that when a processor requests that particular data it will be hit in the L1 cache and which improves the overall performance.

And typically most of the scientific applications you can see that there are so many loops and compiler can optimize these loops efficiently. Also we can consider the blocking mechanism. So, consider a matrix multiplication. So, in this the elements of matrix z are repeatedly accessed and then the elements of the other 2 matrices. And in this particular example we consider colour coding where the dark shaded blocks indicates that these are the elements in the matrix, which are recently accessed and the light shaded ones are accessed but not so recently.

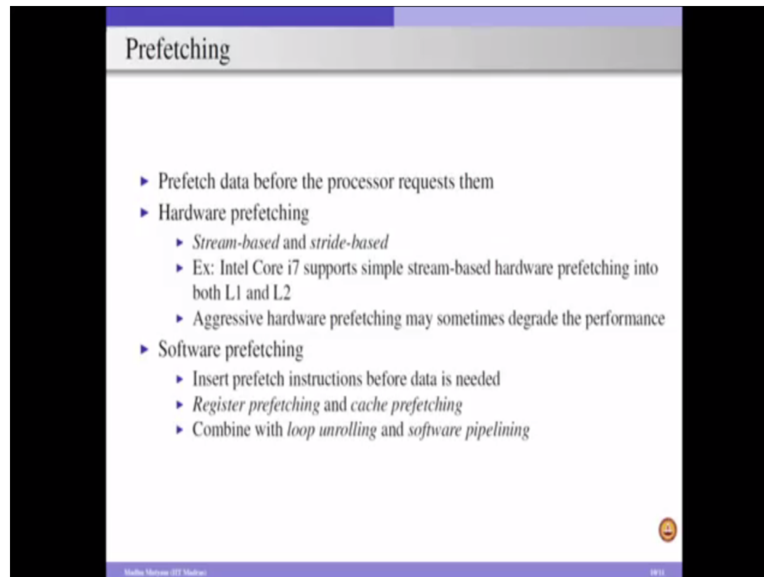
And white blocks indicate that these are the elements that are not accessed at all in those corresponding matrices. So, effectively if I consider a normal matrix multiplication then our accesses will be all over the arrays. And as a result caches cannot the exploit the locality. And the cache space cannot be efficiently utilized to perform these operations, but whereas, if consider the matrix multiplication in a block matrix multiplication method, where we divide the matrices into smaller blocks and we perform the multiplications on those things.

And if we do that, then our access patterns will be confined to smaller regions in the memory. So that these accessed elements can be efficiently kept in the cache memory which can improve the locality and cache performance can be improved significantly. And finally, we can consider a prefetching method, what is Prefetch? Prefetch is fetch the data before the processor requests that particular data. Typically whenever processor wants the data it issues a load request to the cache.

So, that the cache will look at that address and search in that cache with that particular address and if the data is available in the cache. So, it supplies the data to the processor. If the data is not there for that particular address location in the cache, it goes to the next level and which may take several cycles. And meanwhile processor is sitting idle. To reduce this penalty we can actually issue these load requests early in advance, but for that we need to predict which are the load request processor is going to generate in future. So, for that we can take the help of hardware or software. Effectively, we can come up with hardware prefetching

techniques or software prefetching techniques. In the case of hardware prefetching techniques.

(Refer Slide Time: 38:20)



We can consider stream based prefetching or stride based prefetching. We provide extra hardware with the processor and this hardware keeps track of what is the access pattern that happens for the given application, when it is executing on the processor. And these access patterns can exhibit stream based accesses or stride based accesses. Depending upon the aggressiveness of the prefetching it can either prefetch one block or it can prefetch multiple blocks and so on. So, typically this prefetching happens at a granularity of cache block.

When there is a miss happens for one particular block, if it is stream based prefetching it prefetches subsequent blocks to that particular address. In the case of stride based prefetching, sometimes applications access not the contiguous locations, but with a fixed stride distance. In the stride based prefetching is typically the access patterns are following a particular stride, a stride of 10 bytes, a stride of 20 bytes, a stride of 100 bytes or something.

So, our hardware prefetching unit captures this stride difference and whenever there is a miss to particular address block, this hardware prefetching unit which is implementing stride based prefetching, adds the stride difference to the previously missed address. And it prefetches the corresponding block into the cache. So, that when the processor actually requests the data the data is available in the cache. So, as long as the prefetching method is working fine, processor can improve performance significantly because all processor requests can get a hit

in the cache memory, but too much aggressiveness applied in the prefetching can result into performance penalty also.

Effectively, if you are prefetching logic is not efficient, this is going to pollute the cache which is in turn increases overall miss rate. So, as a result we need to come up with efficient prefetching methods, such that there is no significant pollution in the cache. And also the processor when it requests the data the data is available. So, without increasing the conflict miss rate of the useful data we have to keep our prefetched blocks into the cache.

And also another reason, another thing we have to consider with the prefetching logic is because the prefetched request can be competing with actual demand request from the processor, when we are accessing the L2 cache memory and so on. So, again we have to give priority for a demand request, as processor is waiting for these demand request to be services as early as possible than servicing the prefetch requests.

Another reason is we are not sure whether the prefetch request may be actually required by the processor in the future and so on. So, we have to keep all these things into consideration when we are dealing with the prefetching logic. And the current processors have these hardware prefetchers at L1 and L2 and because these prefetching can sometimes degrade the performance. So, these processors also provide a mechanism to turn off this prefetching.

So, we can turn off prefetching and then execute in a normal way. If the applications are going to exploit the benefits of these prefetching, then we can turn on in those cases. So, this is about the hardware prefetching, but we also have software prefetching, where the compiler can insert these prefetched requests by looking at the access patterns. So, the advantage with the software prefetching is, we do not have to incur extra hardware, but the disadvantage is it is going to increase your instruction count.

So, once we have this software prefetching, we can actually apply this software prefetching either for prefetching the data into the cache or prefetching the data into a register. If I am prefetching the data into a register it is called register prefetching, if I am prefetching data into a cache it is called as a cache prefetching. And we can also apply the software prefetching along with other compiler optimizations, such as loop unrolling and software pipeline. Loop unrolling is a concept where a loop is unrolled.

For example, if I have a for loop,

$$For(i=0; i<100; i=i+1)$$
$$A[i]=B[i]*C[i]$$

So, what I can do is, I can unroll the loop by 5 times such that my for loop will be,

$$For(i=0; i<100; i+=5)$$
$$\{$$
$$A[i]=B[i]*C[i]$$
$$A[i+1]=B[i+1]*C[i+1]$$
$$\ldots$$
$$A[i+4]=B[i+4]*C[i+4]$$
$$\}$$

Effectively previously we considered one instruction in the body of the for loop, in the original the loop.

When we unroll the loop 5 times, we inserted 5 instructions in the body of the for loop and, reduce the, adjust the loop iterations accordingly. And we can exploit the cache benefits when we apply this loop unrolling. So, these are the simple technique typically most of the current day compilers apply to improve the overall performance in executing the loops. And in addition to that we can also apply software pipelining.

The software pipelining is similar to a hardware pipelining, where for example, the loop body of a for loop has a dependency in executing the instructions. Now, when we apply software pipelining, we can unroll the loop to a certain extent and then exploit the independent instructions in this extended loop body. So, that once we have independent instructions we can perform these operations simultaneously. So, this is the simple concept considered in compilers as an optimization. So, we can apply this software prefetching mechanism in addition to these loop controlling and software pipelining to improve the overall performance. So, with this I am concluding this cache unit section.

Thank you.