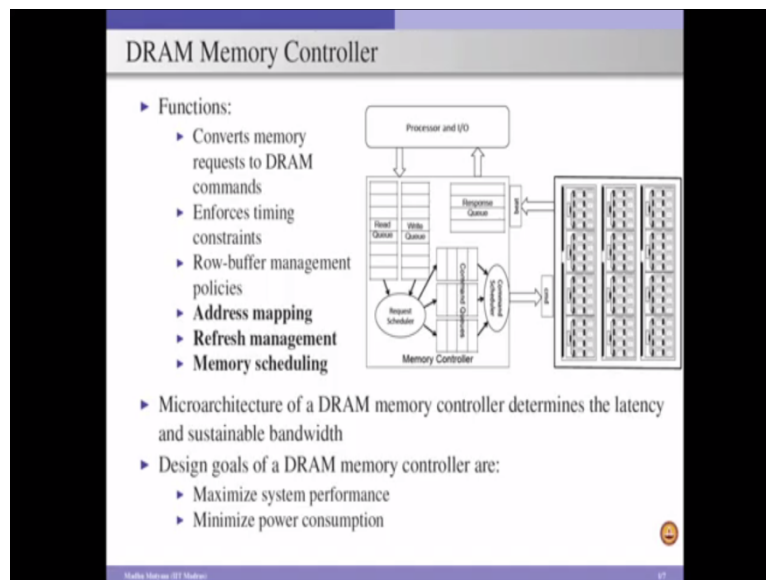


**Computer Architecture**  
**Prof. Madhu Mutyam**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras.**

**Module – 04**  
**Lecture – 12**  
**Memory Hierarchy Design (Part 7)**

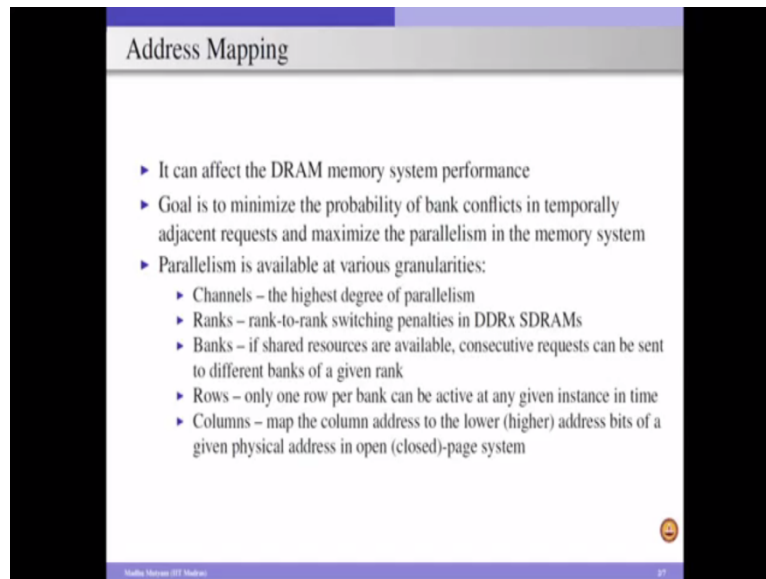
So, in the last module, we introduced memory controller design and discussed three important functions of memory controller design. Now in this module, we will continue with the remaining three important functions.

(Refer Time Slide: 00:28)



So, those are address mapping mechanisms, refresh management techniques, and the memory scheduling. So, we know that processor issues load, and store requests in terms of physical address to the memory and but actually the data stored in the memory in terms of the channels, ranks, banks, rows and so on because the physical layout of the memory consists of the set of DRAM devices, part of the DIMMs and logically this entire DRAM memory can be considered as set of rows, set of banks, set of ranks and set of channels. Effectively, the address mapping is a mechanism which resolves the physical address issued by the processor to the memory in terms of the channel id, rank id, bank id, row id and column id. And, why do we have to study this address mapping mechanisms?

(Refer Time Slide: 01:28)



The address mapping mechanism is a key function in the DRAM memory controller design, because it affects the performance significantly. Consider a scenario, where if you are going to map the consecutively addressed cache lines to the same row in a bank, then when the first request comes through the DRAM device, we activate the row so that, the subsequent requests can get a hit from the opened row.

So, as a result we can exploit the row buffer locality and that improves the performance. On the other hand, For example, if you are going to place these consecutive addressed cache lines to two different rows in the same bank. Now, when we open the first row to service the first request, after servicing that, now when we want to go for the second request, we have to close this, because there is a row conflict. So, this shows that if the address mapping is not proper, we may incur significant number of conflict misses at the row level, row buffer level, and that degrades the performance. And similarly, so if the address mapping is not proper, it may so happen that the all these addresses can go to the different ranks, and when we want to move from one rank to another within a channel then we need to incur rank to rank switching.

So, as a result we have to ensure that proper address mapping mechanism is designed for our memory controller to work efficiently, and to improve the overall performance. Not only performance, it also impacts the overall power consumption. So, the goal of any address mapping mechanism is to minimize the chances of bank conflicts, in temporally adjacent requests and maximize the parallelism in the overall memory system.

Once we have a good address mapping, which exploits parallelism available at different levels, then the overall performance can be improved. And similarly, once we minimize the bank conflicts for our temporally adjacent requests so that, the performance can be improved. So, before discussing the address mapping mechanism, let us see, so what are the granularities at which a parallelism can be available in the memory system.

It is available at different levels. We know that a DRAM memory system consists of channels, which in turn consists of collection of ranks. Each rank consists of collection of banks, and each bank consists of set of rows, and a row consists of collection of columns. So, at the highest level we have these channels.

And so, once we map our consecutively addressed blocks to the different channels then when we want to access these blocks, we can send the requests simultaneously to these two channels, or whatever the available number of channels, and we can exploit the parallelism significantly. So, that means like always the blocks, which are consecutive in access, if you are going to distribute that to multiple channels and all these channels can work independently and we can improve the performance significantly.

So, that indicates that, when we are coming up with an address mapping, we have to ensure that the channel id in the address part should be mapped to the lower order bits in the address. The next one is a rank, we know that a single channel can cover multiple ranks, and if you are going to distribute our consecutively addressed blocks to multiple ranks, now when we are accessing one rank for a particular cache block, and now we want to access consecutively addressed blocks, then we have to go to the other rank.

If you do that, we need to incur a penalty of rank to rank switching  $t_{RRS}$ , and we discussed in the last module, for a DDR4 system this rank to rank switching is 2 cycles, and we unnecessarily incur a penalty of 2 cycles, if we are distributing our consecutive address blocks, to multiple ranks. So, as a result we need to keep the consecutively addressed blocks, as much as possible to a single rank. So, that we do not have to incur this rank to rank switching.

We know that each rank consists of multiple banks. Now, all these blocks, all these banks within a rank can work independently. So, as a result what we can do is, we can map our consecutively addressed blocks to different banks within a rank. So, of course, when we want to access consecutive addressed blocks, we need to switch between bank to bank. But unlike

rank to rank delay, we do not have any bank to bank delay, if the banks are within a particular rank.

So, as a result, we can exploit parallelism by accessing multiple banks that are there inside a single rank and this parallelization can be exploited as long as we have the shared resources available. The shared resources such as the address bus, command bus, data bus and so on. So, that way we can exploit parallelism at the bank level. Now, within a bank, we will have large collection of rows and now once we have large collection of rows, we can distribute our data.

But, if you are going to distribute our consecutively addressed cache blocks to different rows in a single bank then we are going to incur a penalty. Because, in a bank even if there are 100 or 200 or thousands of rows, at any point of time, we can work with only 1 row. So, when we open a particular row, then we cannot access any other rows in the bank, unless we close the previously opened row.

So, that says that, if we map our consecutively addressed cache blocks to different rows within a bank, we incur a significant penalty in terms of row closing and row activation. So, we will incur  $t_{PRE}$  and then  $t_{ACT}$ . On the other hand, if you are going to map our consecutively addressed cache blocks to a single row within a bank, we can exploit the row buffer locality significantly. In such case, we incur only a  $t_{CAS}$  time to access the data from the opened page.

So, that indicates that we have to map our row id to the higher order bits in our address. And finally, the columns where do, we map our columns, the column is going to specify the particular position in our opened row, to read the data. Now, again in this if you are going to use an open page policy, we already discussed page management policies in the last module. An open page policy, a page will be opened until there is a row conflict, and whereas, in the case of closed page policy as soon as we perform a CAS, we start issuing a PRE. So, that we can close this page, and we can make the page status to be empty. So that, when we want to access the next row, we can start with ACT command, so that, the PRE will not be in the critical path.

Now, when I consider an open page policy, because the opened row will be there as it is until there is a conflict, so as a result if we are considering an open page policy, we have to map our column id bits towards the lower order bits of our address. And since the lower order bits

will change frequently and consecutive blocks can get a hit in the opened page. Whereas, in the case of closed page policy, because we are going to close immediately after servicing a particular request.

So, we need to map our column id bits to the higher order bits of the address. So, this way like we can exploit our parallelization at different levels and we need to map our channel id, rank id, bank id, row id and column id to appropriate positions in our given address. Now, we will consider an example to see how to come up with a decent address mapping mechanism for open page policy and closed page policy.

(Refer Time Slide: 11:16)

**Baseline Address Mapping Schemes**

- ▶ Let the total size of the memory be  $K \times L \times B \times R \times C \times V$ , where
  - ▶  $K$ : # of channels ( $= 2^k$ )
  - ▶  $L$ : # of ranks per channels ( $= 2^l$ )
  - ▶  $B$ : # of banks per rank ( $= 2^b$ )
  - ▶  $R$ : # of rows per bank ( $= 2^r$ )
  - ▶  $C$ : # of columns per row ( $= 2^c$ )
  - ▶  $V$ : # of bytes per column ( $= 2^v$ )
- ▶ The number of bytes per row ( $C \times V$ ) can be expressed as  $N \times Z$ , where
  - ▶  $N$ : # of cache lines per row ( $= 2^n$ )
  - ▶  $Z$ : # of bytes per cache line ( $= 2^z$ )
- ▶ The baseline open-page address mapping scheme is  $r : l : b : n : k : z$
- ▶ The baseline closed-page address mapping scheme is  $r : n : l : b : k : z$

	$B_0$	$B_1$	$B_2$	$B_3$	$B_4$	$B_5$	$B_6$	$B_7$
$C_0$	0	8	2	10	4	12	6	14
$R_1$	16	24	18	26	20	28	22	30

	$B_0$	$B_1$	$B_2$	$B_3$	$B_4$	$B_5$	$B_6$	$B_7$
$C_0$	0	2	8	10	16	18	24	26
$R_1$	4	6	12	14	20	22	28	30

$U : l : b : n : k : z$                        $U : n : l : b : k : z$

So, consider a memory system with  $K$  channels, each channel having the  $L$  ranks per channel, and each rank has  $B$  banks in it, each bank has  $R$  rows in it, and each row has  $C$  columns, and each column has  $V$  bytes in it. Effectively, the overall size of the memory is the product of  $K$ ,  $L$ ,  $B$ ,  $R$ ,  $C$  and  $V$  bytes. And, we also assume that,

$$K = 2^k$$

$$L = 2^l$$

$$B = 2^b$$

Effectively if we have four channels in our memory, we require 2 bits of information to code the channel id. Similarly, for example, if we have 8 banks per rank, then we need 3 bit bank id and so on. Now, once we have the total memory of size  $K * L * B * R * C * V$  bytes, we require now. Because, typically the data transfer between the multiple levels of caches, and

the memory happens at the cache line granularity. So, we relate this last two terms  $C$  and  $V$  in terms of the cache line.

So, for that let us assume that,  $N$  be the number of the cache lines per row, and  $Z$  be the bytes per cache line, so that,  $C * V$  can be replaced by  $N * Z$ . so, effectively we have,

$$\text{Total memory size} = K * L * B * R * N * Z$$

and this memory can be expressed using  $K + L + B + R + N + Z$  number of bits, where each of these are the number of bits required to indicate the corresponding component.

So, if I consider a base line name address mapping mechanism, I can consider the higher order bits of my address pointing to the channel id. Next few bits are pointing to the rank id, next few bits are pointing to the bank id, next few bits are pointing to the row within the bank, next few bits are going to specify the cache line id within the row, and finally, the block offset inside a cache line.

So, when we consider this naive address mapping what is going to happen is because, our channel id is at the higher order bits. As, a result all our addresses are confined to one channel, and if the application is requiring more space than the space covered by channel, then only it will go to the second channel. So, that indicates that this address mapping cannot exploit the parallelism at the channel level.

When we are considering, because in this naive address mapping, our, the channel, the cache line id is mapped to the lower order bits. And, if you're using a closed page policy, and if the application is accessing subsequent cache lines so for every subsequent cache line access, we incur a penalty in terms of closing the row and then opening the row. So, that means in a closed page policy this name address mapping may not be efficient.

Similarly, whether it is a closed page or open page policy because our channel id is mapped to the higher order bits, we cannot exploit, we cannot exploit the parallelization at the channel level. So, that is the reason why this naive address mapping mechanism where the channel id is at the MSB bits and whereas, the cache line id is almost to the close to the lower order bits is not a good idea.

We need to come up with efficient mechanisms for different page management policies. And remember in our page management policies, we consider three different techniques one is an

open page policy, the other one is a closed page policy, and the last one is a hybrid policy. And we already discussed that, the hybrid policy is efficient in terms of exploiting the access behavior or access rates.

So, it is effectively the hybrid policy can dynamically adjust itself, based on the access rates of the request coming to the memory, and also looks at the access locality within a rank. So, among these three policies the hybrid policy is better. But, the same thing cannot be applied in the case of address mapping and when we stick to one address mapping, we have to stick to that throughout. So, as a result we give address mappings only for either, the open page policy or closed page policy.

We cannot give that for a hybrid policy. So, when we consider a base line address mapping mechanism for open page policy, the address mapping looks something like this,  $r : l : b : n : k : z$ , where  $z$  is the byte address in a cache line and  $k$  is the channel id,  $n$  is the cache line id,  $b$  is the bank id,  $l$  is the rank id and finally,  $r$  is the row id.

So row is mapped to the MSB, the change in the bit values in the MSB position is much lower compared to the change in the bit values, which are at the lower order positions. So, as a result once we map our row id to the MSB position, so, it may not change frequently. As a result once we open a row, the row can be opened until there is a conflict to that particular row, and we can minimize the row conflicts, one by mapping our row to the MSB position. And since,  $k$  is mapped towards the LSB.

So, we can interleave cache lines across multiple channels. Let us assume that, we have 2 channels and so that the first cache line will be stored in one channel, second cache line is stored in the second channel, and this continues alternatively, and after the channel id we get the cache line id. So, as a result this shows that the consecutive cache lines will go to the subsequent channels and after the channel id we have a bank id.

So, this says like for the same rank and the same bank the cache lines belong to a same row, can be mapped on to multiple channels. So that, when we open a row from a bank of a rank, the row can be opened for significant amount of time to get more and more row hits, and in the case of closed page policy, we know that once a request is serviced we have to close the page.

Now, in this case if you are going to map our adjacent cache lines to the same row we are going to unnecessarily incur a penalty. Because, the closed page policy clearly says that, once the request is done, the opened page needs to be closed. So, that indicates that we have to move our the column id towards the MSB side. So, when we consider that, we can clearly see that, the base line address mapping for closed page policy is something like this, where  $r$  is still at the MSB position as that of the open page addressing mapping.

Similarly,  $k$  is also towards the LSB side as that of the open page policy. But, the channel the column id is moved towards the MSB positions, unlike the open page policy where it is towards the LSB side. So, this indicates that the columns of a same row should be distributed to multiple channels. But, within a channel it should be distributed to different ranks and different banks.

But, whereas, in the case of open page policy we want to distribute the cache lines to different channels but, within a channel we want to map the same row, same rank and same bank. So, that is the whole difference between open page and the closed page policies, open page and the closed page address mapping schemes. And now, we will consider a simple example, where we have a memory system consists of 2 channels, and each channel has 2 ranks, and each rank has 2 banks, and each bank has multiple rows.

Now, we will consider, also we consider, per row we will have 4 cache lines. When we consider 32 consecutive cache lines and if we consider an open page policy, we can clearly see that, the consecutive addressed cache lines are mapped to channel 0 and channel 1, 0 is mapped to channel 0, 1 is mapped to channel 1. Similarly, cache line 2 is mapped to channel 0, and 3 is mapped to channel 1 and so on. And within that, again for bank 0 of rank 0 of channel 0, if we see, the cache line 0, 2, 4 and 6 are mapped and similarly, for channel 1, rank 0 and bank 0, we map the other 4 that is block number 1, 3, 5 and 7. And similarly, we can expand for the other block positions in the address mapping for open page policies.

When we consider, the base line closed page address mapping mechanism, we can clearly see. So, if we consider the first 8 blocks, the cache line 0 to cache line 7, cache line 0 is mapped to channel 0, rank 0, and bank 0, and cache line 1 is mapped to channel 1, rank 0, bank 0, and cache line 2 is mapped to channel 0, rank 0, bank 1, and the cache line 3 is mapped to channel 1, rank 0, bank 1 and so on.



So, effectively first 8 blocks, if we consider, these 8 blocks are covering all the channels, all the ranks, and all the banks in the case of closed page address mapping mechanism. But, whereas in the case of open page policy the first 8 cache lines are confined only to a single rank, within that single rank, it is confined to only a single bank. But, these blocks are distributed across 2 channels.

So, that way like because the objective of the working of open page policy is different from the closed page policy and the objective of open page policy is different from that of the closed page policy. So, when we consider address mappings, also we need to consider these page management policies, and need to come up with efficient address mapping mechanisms, by considering the underlying page management policies. So, with this I am concluding this address mapping mechanism, and now we will move on to the refresh mechanisms.

(Refer Time Slide: 23:10)

### Refresh Management

- ▶ Read data and restore it in DRAM devices, before the worst-case data decay time
- ▶ Consumes available bandwidth and power
- ▶ Goal: minimize controller complexity or bandwidth impact or power consumption
- ▶ *All-bank concurrent refresh*: a single refresh command to the DRAM device
  - ▶ *Refresh address register* to store the address of the last refreshed row
- ▶ Memory controller issues 8192 refresh commands to the DRAM device for every 64ms
- ▶ *Distributed refresh*:

The diagram illustrates the timing of refresh commands. A horizontal timeline shows a series of refresh commands. The first command is labeled '1st REF cmd.' and the last is '8192th REF cmd.'. A double-headed arrow below the timeline indicates a 64ms interval between the first and last commands. A red bar below the first command is labeled 'tRFC'. A double-headed arrow below the timeline indicates a period of 'tREFI = 7.8us' between the first and last commands. A small smiley face icon is visible in the bottom right corner of the slide.

We already discussed that, the charge in the DRAM cells are going to leak gradually and we need to refresh these DRAM cells to maintain the data integrity. So, as a result, we need to refresh these cells at regular intervals. And once we do this, automatically the corresponding DRAM device cannot be available, for servicing any request from the processors or I/O devices and that is going to degrade the performance. So, as a result we need to understand how the refresh mechanism is happening.

So that, we can know what is the overhead we will get in terms of the overall performance, and then we can think of coming up with an efficient refresh management policies. So, we

know that, the data in the DRAM cells decay, based on the temperature on the DRAM device. If the temperature is below 85 degree centigrade, then the DRAM cell can retain the data for up to 64 milliseconds but if the temperature is above 85 degrees then it is going to retain the data only for 32 milliseconds.

So, within that time interval, we need to refresh the DRAM cells so that, it retains the data for next 64 millisecond or 32 milliseconds based on the temperature on the DRAM device. So, since these refresh operations are interfering with the request from the processor, and the I/O devices, and the DRAM device is not available during this refresh interval. So, that indicates that, it is going to have an impact on the bandwidth and also it is going to consume the power.

So, as a result the refresh mechanism is important thing to deal with and we need to come up with an efficient mechanism to minimize the overhead associated with this bandwidth loss and the power consumption. In order to minimize the impact of the refresh on the bandwidth or the power consumption or in order to come up with a simple memory controller design, we need to come up with an efficient refresh mechanism.

There are several refresh management techniques proposed in the literature, but we are going to discuss the basic refresh management techniques in this module. In order to reduce the complexity associated with the DRAM memory controller design, one simple technique we can do is, we will go for all bank concurrent refresh mechanism. So, in this mechanism, the memory controller issues a single refresh command to a DRAM device and once a command, refresh command are given to the DRAM device all the banks inside the DRAM device will undergo for refresh and they refresh a single row in each of these DRAM banks. Let us assume that, if a device has 8 banks, when I issue a refresh command so, from each bank we are going to refresh 1 row though we have not discussed, one important timing parameter associated with the DRAM memory controller, that is  $t_{FAW}$  4 activation window.

This timing parameter is given mainly to ensure that the peak power consumption of this memory system is not crossing a threshold value, this  $t_{FAW}$  indicates that within an interval, we should not activate more than 4 rows. So, when a device has 8 DRAM banks, when I issue a refresh command, I cannot refresh 1 row from all these 8 banks simultaneously. We have to refresh 4 rows from, some 4 banks inside this DRAM device and wait for some time and then refresh the other 4 from the remaining banks.

And this is mainly to satisfy the  $t_{FAW}$  constraint. Now, this is simple, the all bank concurrent refresh mechanism is simple, in terms of memory controller design, the reason is all we require is one refresh command so that, once the command is given the refresh operation is performed. And to implement this, all we require is, we need one register that is called as a refresh address register, which stores the address of the last refreshed row.

So, whenever the memory controller issues a refresh command, it updates the refresh address register with that row id, and once that is done, again it looks at the refresh address register contents and then increments and then issues the refresh to the subsequent row. Effectively, if we have a DRAM bank consists of 8192 rows, is a standard typically, most of the low capacity DRAM devices will have 8192 rows per DRAM bank. In such a scenario, when I apply this all bank concurrent refresh, first I start with row 0, I refresh it then go to row 1 refresh it, and then go to row 2 and so on. We will continue to the last row of the bank and because a refresh command is given to all the banks within a device. So, finally at the end, once we issue a refresh command to the last row of each of these banks inside a DRAM device the entire DRAM device is said to be refreshed.

But, if we do this it is going to incur significant penalty if processor requests are waiting to be serviced during this interval. We know that each refresh operation consists of reading a row, activating a row, reading the row content to the sense amplifiers and writing the data from sense amplifiers back to the row.

So, this we need to perform for each of the rows inside a DRAM bank and when we have a DRAM bank of 8192 rows, then it is going to take significant amount of time and that is going to degrade the performance significantly. And especially when there is a performance critical request from the processor is waiting to be serviced by the memory and if the memory is busy with the refresh, then we are going to incur significant penalty.

So, though this all bank concurrent request is efficient in terms of the design complexity but, it is not efficient from the performance point of view. And the JEDEC standard specifies that in a DRAM device, we need to issue a total of 8192 refresh commands, so that, for one refresh command we can refresh one row from a bank inside the DRAM device. But, now these days, we are getting the DRAM DIMMs with higher and higher capacity, such as, For example, we have a DIMM with 32 gigabits of capacity.

So, as we want to increase the capacity of DRAM device, we have to increase the number of rows inside the DRAM bank. Once, we increase the number of rows so effectively for each refresh command, we have to refresh not just 1 row but multiple rows from each bank. In other words, for example, if we have a DRAM bank consists of 32k rows so for each refresh command, we have to refresh 4 rows and that is going to take significant amount of time for each refresh operation.

Effectively, our  $t_{RFC}$  is going to increase, because our  $t_{RFC}$  is proportional to the number of rows to be refreshed per the refreshed command. So, as a result we are going to incur a significant penalty. So, that means like, all bank concurrent refresh, when we are doing contiguously without any interrupt, without any gap between any refreshes then we are going to incur a penalty significantly.

In order to overcome this, we can come up with a different mechanism called as distributed refresh. This distributed refresh mechanism says that, rather than refreshing all the rows inside a bank so we, what we can do is, we will refresh few number of rows now, and then give a gap so that, processor request if any waiting to be serviced will be serviced. And, then again move on to a refresh, and then again back to a normal mode, and then to a refresh mode.

So, effectively we interleave the refresh mode and the normal mode of operations so that, rather than issuing all 8192 refresh commands continuously, we give one refresh command now, wait for some amount of time to service processor or I/O request and then again go for the second refresh command and then again wait for some amount of time to service processor or I/O request and this continues further. And the interval between two refresh commands is called as  $t_{refresh}$  interval,  $t_{REFI}$ . And, this is effectively 7.8 microseconds where we get this value by dividing the 64 millisecond interval with 8192, and if it is in extended temperature mode, then our interval will be 32 milliseconds. So, as a result  $t_{REFI}$  will be half of the 7.8 microseconds. So, the main idea of this distributed refresh is do not refresh all the rows, do not issue all the refresh commands to a DRAM device continuously, give a refresh command 1 and then give some space so that the processor request can be serviced then again go for a second refresh command, and so on.

Interleave a refresh and the processor operations simultaneously, so that, the performance penalty due to refresh can be minimized. So, this is about the refresh management and finally

we will move on to memory scheduling. We already seen, from the microarchitecture of memory controller that there are two schedulers, one is for a request scheduler, the other one is a command scheduler. And the request scheduler is actually, taking the request from the request queues. And we consider two request queues, one for storing read request from the processor and I/O devices, and the other one is to store the write request.

(Refer Time Slide: 34:47)

Memory Scheduling

- ▶ Scheduling at request-level and command-level
- ▶ Two separate queues to hold reads and writes
  - ▶ Memory controller works in *read major mode* and *write major mode*
- ▶ Request scheduler can select requests in out-of-order from the request queues
- ▶ Per-bank command queues can be considered
- ▶ Command scheduler selects a command queue based on *Round-Robin*, *First Ready First Serve (FRFS)*, or *age-based* policies
- ▶ Command scheduler picks the commands from command queue in FIFO order

So, in the case of a request scheduling, the scheduler can pick one read request, and then go to a write request, and then go a read request, and so on. It can interleave between read and write, but if we do so, we unnecessarily incur a penalty, because as we discussed in the last module, whenever we are moving from a write operation to a read operation, we need to release the I/O gating resources, held by the previous write operation.

So that, the subsequent read operation can perform its operation successfully. In order to do that, we will incur a penalty of 4 cycles that is  $t_{\text{write to read delay } t_{\text{WTR}}$ . So, as a result it is not a good idea to select requests between the read queues, write queue in an alternate cycle. We have to consider a different mechanism, wherein typically we continuously get the request, select the request from a particular queue, may be from the read queue and service the request to a certain point, where the number of request in the write queue is crossing a higher danger mark. If it crossing a higher danger marks, that indicates that the write queue sooner going to be full, and then we have to stall everything and then drain the request from the write queue. That is going to degrade the performance significantly.

And as a result, whenever the number of entries in the write queue is going to cross a danger limit, a higher danger limit, then we will move on to the writes, and then we will pick the request from the write queue. So, while we are picking the request from the write queue, then the request in the read queue will keep on increasing because, the processor or I/O device can send the request, and this is going to go up and while we are servicing the request from the write queue, after some time, the number of request in the write queue will go to a lower value.

So, it is like once it crosses the lower threshold, then immediately we will move on to the read queue and select the request. So that, again it takes some more time for write queue, to fill in its entries, and then we will service the read request from the read queue. So, whenever, we are performing operations or whenever we are selecting request from a read queue, then we call it is as a read major mode.

If we are going to select the request from the write queue, then we are going to call it as write major mode. Similarly, like the lower and the upper thresholds for write queue, we also have the lower and upper thresholds for the read queue. Once these thresholds are reached, then automatically we are going to move on to the other queue, and then select the request continuously from that particular thing.

So, with that like we are going to select a request from a read queue or a write queue based on whether, it is a read major mode or a write major mode. And once we select a request, now we will convert that request into memory commands. By the way, when we are selecting a request from a read queue or from a write queue, our request scheduler can reorder the request. So, as a result our request queue, request scheduler can select request in out of order fashion.

It, it needs not follow the same order in which the processor initiated the request and so on. Because, we know that the request to the request queues, read queue or write queue can come from different agents. The requests may be from a display agent, request may be from a GPU graphic agent, request may be from a CPU agents, or we can even CPU agent also, it may be from different applications.

So, we can reorder these requests based on the application criticality or based on the application importance or the agent importance. So, the request scheduler will do these reorderings and select a particular request from a particular queue based on read major mode or

a write major mode. And then, the selected request will be converted into set of memory commands. And once the request is selected and converted into set of memory commands and these memory commands will be stored in a bank wise command queues.

We already discussed in the previous module that so, we have a collection of command queues which are at a bank granularity. So, each bank will have one queue. Of course it is not a queue effectively because we perform the operations in the FIFO order. Of course, the advanced memory controller designs will go for different orderings within the bank. But if you are going to do this out of ordering of request selection from the command selection, from the command queues.

Then, so we have to insert PRE, ACT and CAS for each of the requests. But, if we are going to go for a FIFO ordering in the command queues then context aware conversion of a request to the commands can be applicable. So, in the last module, we already discussed like, if there is second request to the same row as that of that of the previous request and then I can issue only one command that is a CAS to the corresponding command queue.

But, if I am going to go for an out of order execution and the out of order selection of the command queues at the bank level, then for each request, we have to pick up 3 commands PRE, ACT and CAS and based on the context at the state of the row buffer at that particular position, we can discard some of the commands. But, that is going to make the overall design complex and we are not going to consider that in our module.

To make the memory controller design simple, we will consider our command queues work in FIFO order. The order in which the commands are inserted into the command queues and these command queues are going to work at the bank level, and once we insert some set of commands to a particular bank level command queue, and the request in that queue will be processed in the order in which they are inserted.

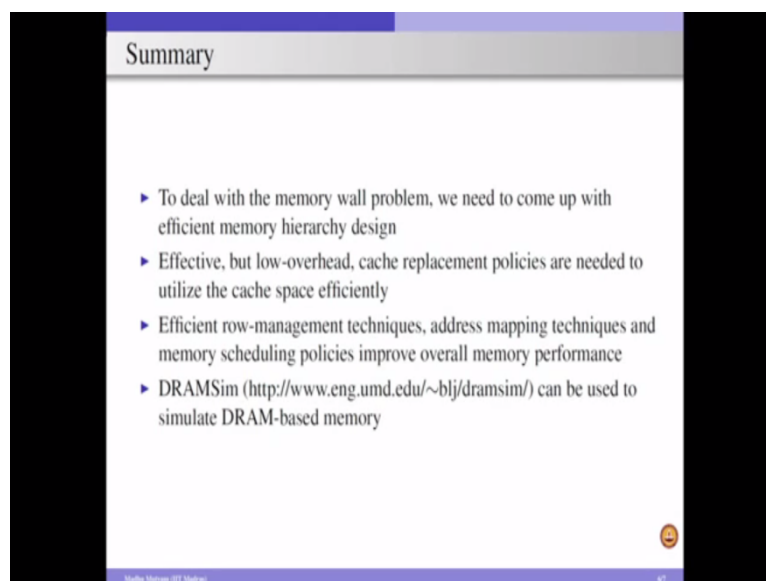
The first request will go first, and the second request will go second, and once we have the request converted into commands and the commands are stored in bank level command queues now, there is a command scheduler and the role of the command scheduler is to select one particular command queue from the available lot of the command queues. And for that, it can go for different policies, one may be a simple round robin policy, this is mainly for to ensure the fairness among multiple the banks.

So, just go for a bank 0, go to the bank 0 command queue, next bank 1 command queue bank 2, 3 and so on and come back to the bank 0 within a rank and similarly, it can select across the ranks and the channels. This is a simple design round robin, but this may not give you performance improvement significantly. And similarly, there is another policy, which is first ready first serve.

Whenever we want to select a request, whenever we want to select a command, we can just look at all the first entries in all the command queues and whichever is ready, among all the ready commands, we will see which one is issued first. So that we can take care of the age into consideration so that, some agents may not be starved to service its request and so on.

So, this is again like you can improve performance as well as fairness can be improved here. And similarly, you can also consider other age based policies, in selecting a particular, the command queue among all the available command queues, by using these commands scheduler. So, with that we will complete this, the memory hierarchy's design.

(Refer Time Slide: 43:10)



Summary

- ▶ To deal with the memory wall problem, we need to come up with efficient memory hierarchy design
- ▶ Effective, but low-overhead, cache replacement policies are needed to utilize the cache space efficiently
- ▶ Efficient row-management techniques, address mapping techniques and memory scheduling policies improve overall memory performance
- ▶ DRAMSim (<http://www.eng.umd.edu/~blj/dramsim/>) can be used to simulate DRAM-based memory

Nadim Mithan (IT Madras) 47

Summary, in the last, the two weeks of lectures, we considered a cache memory design, the internal organization of the cache memory and the several optimizations that can be performed in cache memories to improve performance and minimize energy consumption.

We also discussed the microarchitecture of the memory controller, we discussed the basics of DRAM based memory and what are the issues involved in designing, what are the



functionalities of memory controller and we discussed page management policies, address mapping mechanisms, refresh mechanisms.

In order to come up with efficient memory hierarchy design, we need to come up with efficient cache replacement policies, which may be adapted to the application behaviors. And at the same time we should not incur significant overhead. And also from the memory, DRAM based memory point of view, we need to come up with an efficient address mechanism, efficient page management policies, as well as efficient memory scheduling policies.

And those who are interested in understanding more about the functioning of DRAM based memory, they can look at DRAMSim, which is a publicly available DRAM simulator from University Of Maryland and you can download this and then you can simulate DRAM based memory, and you can see the impact of the different address mapping mechanisms, or you can see for the different configurations of the memory, increasing the channels, increasing the ranks or considering a different address mapping mechanisms or applying the refresh management techniques and several other things. You can play with the simulator. So with that I am concluding this memory hierarchy design.

Thank you.