1)

a) Locality of reference or commonly known as the principle of locality simplify states that most of the programs tend to reuse the already executed instruction and/or data they have accessed before.

In general with respect to instructions, it can be assumed that almost all the programs spend about 90% of the total execution time, executing a single block of code. This property/principle can be used to predict the pattern of accesses thus reducing the execution time.

For data accesses, this type of locality/prediction holds true but it is not as strong as for code/instruction accesses.

Thus, locality of reference can be extensively used for predicting the next instruction using the recent past.

b) It was predicted by the computer architects that ideally they would want a memory which is quite large to hold all the contents of the running program and the data could be easily retrieved from it without any extra delay. Thus, in order to have faster memories, the principle of locality was used to have faster but smaller hardware depending on the power budget guidelines. This design approach ultimately led to different memory hierarchy design – depending on the speed and size with the general rule being smaller memory being faster than the bigger one. This led to the multi-level memory hierarchy system!

c) As stated above the principle of locality somewhat led to the concept of memory hierarchy. Memory hierarchy simply means using different memories (different in speed and size) at different levels (hierarchy). Typically, the modern processors now have a 3-Level Cache Memory hierarchy apart from the main memory. Each lower level is faster and more expensive and closer to the CPU as compared to the level above it. The further comparison can be –

Level 1 Cache – Closest to the CPU. Generally, small in size (from 8 KB to a maximum of 64KB) with very fast memory access.

Level 2 Cache – The next level of memory hierarchy, it is accessed when the data can't be located in L1-cache. Bigger but slow in memory access as compared to the L1 cache. The typical range of this memory can be from 64KB to 128KB of maximum space.

Level 3 Cache – The size of L3 cache is typically few MBs (2-4MB) with the access time being a bit slow due to increased size.

All the above cache memory are designed using the SRAM (Static Random Access Memory) since this technology doesn't require any refreshing as compared to the DRAM memory design. Also, the design of an SRAM memory cell requires only 6 transistors (usually) and due to the above benefits the access is quick as compared to DRAM. But one thing which stops us from making huge SRAM memories is the fact that they are very costly as compared to DRAMs.

The main memory is quite large (a few GBs) and the access is generally very slow as compared to the caches. The technology used is typically DRAM which some optimizations like Synchronous DRAM (SDRAM) by using a clock signal for easy burst transfers.

d) Code segment –

```
for (I = 0; I < 3; I++)
  for (J = 0; J < 2; J++)
    A[I][J] = B[J][0] + 5;
```

Since the elements within the same row are stored contiguously those would exhibit spatial locality. When we access an array – A[I][J] -> this means we are accessing the Ith row and Jth column.

Therefore, the access to array B would always exhibit spatial locality since we are basically accessing it row by row (B[J][0])

Each and every access within the inner loop would not exhibit any spatial locality for array A since the columns would be getting updated on every iteration (A[I][J] – in the inner loop I would be constant but J would be updated on every iteration of the inner loop).
Hence, the access to the array A wouldn't exhibit any spatial locality due to the inner loop.

e) Comparison of the three cache memory organizations with respect to hardware complexity, flexibility in implementing block replacement algorithms and hit ratio –

i) Direct Mapped Cache –
1) Hardware Complexity – Since in direct mapped cache there is exactly one place for one address that is there is only one block per set, the hardware isn't that complex for this type of cache organization.
2) Flexibility in implementing replacement algorithms – Again, in direct mapped cache there is only one set for each block, the replacement policy doesn't have any choice but to replace the only block – hence this type of cache isn't very flexible in implementing replacement algorithms.
3) Hit Ratio – Hit ratio would depend on the size of the cache and the addresses accessed. Typically, the hit ratio for this type of cache would be low since the for one set there would be only one block and this would lead to more cache misses.

ii) Set Associative Cache –
1) Hardware Complexity – In set associative cache there could n (where n is the number of ways) sets for each block of data. Due to this the logic would be a bit complex since we would need to have n comparators to get the matching tag for the particular index. Hence the hardware would be more and a bit complex as compared to direct mapped cache organization.
2) Flexibility in implementing replacement algorithms – Since there would be n blocks for each index access and if each of the ways are full there would be a need of replacement. This gives the designer to choose between certain sets of algorithms – like FIFO replacement technique, Least Recently used (LRU) replacement policy – thus there is a bit of flexibility on the replacement algorithms which could be implemented.
3) Hit Ratio – Hit ratio would increase as compared to the direct mapped case. Since we can store more data for the same index hence this provides better coverage and low cache misses.

iii) Fully Associative Cache –
1) Hardware Complexity – The hardware required would be more and a bit more complex as compared to the set associative cache. In fully associative cache there is no direct mapping hence data can be mapped anywhere in the cache.
2) Flexibility in implementing replacement algorithms – In this type of cache association there could be a case where we can decide which of the line to replace in case there is less space and thus we can have certain replacement algorithms implemented to improve cache hit rate.
3) Hit Ratio – Hit ratio would improve and quite high for this type of cache organization since there is no mapping and data could be placed anywhere giving full associativity. Due to this there would be effectively less evictions and thus higher hit percentage.

2) Cache Size = 64 KB
Block Size = 16 B
Address = 32 Bits

a) Direct Mapped Cache –
Offset bits = Since each block would be 16 Bytes long, for a byte addressable cache memory we would need bits to map the 16 Byte blocks – Number of bits required = 4
Index bits = Since each block is 16 Bytes wide, the number of sets in the cache would be = 64KB/16B = 4K = 4096 sets – Number of bits required to map these 4096 sets would be = 12
Tag bits – The remaining number of bits would correspond to the tag bits = 32 – 16 = 16 bits

b) 4 – way set associate –
Offset bits = Since each block would be 16 Bytes long, for a byte addressable cache memory we would need bits to map the 16 Byte blocks – Number of bits required = 4
Index bits = Since each block is 16 Bytes wide, the total number of sets in the cache would be = 64KB/16B = 4K = 4096 sets. Now, the cache is 4-way set associative hence the number of sets in each way would be = 4K/4 = 1K = 1024 sets – Number of bits required to map these 1024 sets would be = 10
Tag bits – The remaining number of bits would correspond to the tag bits = 32 – 14 = 18 bits

c) Fully associate –
Offset bits = Since each block would be 16 Bytes long, for a byte addressable cache memory we would need bits to map the 16 Byte blocks – Number of bits required = 4
Index bits = Since the cache is fully associative there won’t be any index bits required. 0 bits
Tag bits – The remaining number of bits would correspond to the tag bits = 32 – 4 = 28 bits

3) Fully associative cache with four 1-word blocks – Size of cache = 4 * 4B = 16B
Block size = 4B – 2 bits Hence there would be 30 tag bits
Replacement policy = LRU
address sequence = 1,2,3,4,1,5,2,3,6,5,4,1,6,2,5,4
Let the address be 32 bits wide –

Initial State of cache –

Block	Tag	Valid
1	X	0
2	X	0
3	X	0
4	X	0

After 1st execution –
Addr: 1 Tag: 0 Offset:1

Block	Tag	Valid
1	0	1
2	X	0
3	X	0
4	X	0

After 2nd execution (will hit block 1) –
Addr: 2 Tag: 0 Offset:2

Block	Tag	Valid
1	0	1
2	0	1
3	X	0
4	X	0

After 3rd execution (will hit block 1) –
Addr: 3 Tag: 0 Offset:3

Block	Tag	Valid
1	0	1
2	X	0
3	X	0
4	X	0

After 4th execution –

Addr: 4 Tag: 1 Offset:0

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 5th execution –

Addr: 1 Tag: 0 Offset:1 (will hit block 1)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 6th execution –

Addr: 5 Tag: 1 Offset:1 (will hit block 2)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 7th execution –

Addr: 2 Tag: 0 Offset:2 (will hit block 1)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 8th execution –

Addr: 3 Tag: 0 Offset:3 (will hit block 1)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 9th execution –

Addr: 6 Tag: 1 Offset:2 (will hit block 2)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 10th execution –

Addr: 5 Tag: 1 Offset:1 (will hit block 2)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 11th execution –

Addr: 4 Tag: 1 Offset:0 (will hit block 2)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 12th execution –

Addr: 1 Tag: 0 Offset:1 (will hit block 1)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 13th execution –

Addr: 6 Tag: 1 Offset:2 (will hit block 2)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 14th execution –
Addr: 2 Tag: 0 Offset:2 (will hit block 1)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 15th execution –
Addr: 5 Tag: 1 Offset:1 (will hit block 2)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

After 16th execution –
Addr: 4 Tag: 1 Offset:0 (will hit block 2)

Block	Tag	Valid
1	0	1
2	1	1
3	X	0
4	X	0

4) Virtual Address = 24 Bits
Physical Memory = 64KB
4KB pages

a) Size of each page table entry in bits -
Bits required for page number = Bits in virtual address – offset bits
Offset bits = 12 (as each page is 4KB in size)
Virtual Page number bits = 24 – 12 = 12 Bits
Size of physical page number = Size of physical address space – offset bits
offset bits = 12
Size of physical address space = 64KB = 16 bits
Size of physical page number = 16 -12 = 4 Bits
Size of page table entry = 12 + 3 + 4 = 19 bits (including valid, dirty and use bit)

b) Number of entries in the page table = 2^{12} = 4K entries (4096)
Size of page table = number of entries * size of each entry
=> 4K * 19/8 B = 9.5 KB
Yes, the physical memory can store this data structure.

c) 4-way set associative TLB
Total of 16 TLB entries
Size of each TLB entry -
Size of physical page number = Size of physical address space – offset bits
offset bits = 12
Size of physical address space = 64KB = 16 bits
Size of physical page number = 16 -12 = 4 Bits
Size of virtual page number = Virtual address bits – (offset + index bits)
16 of total TLB entries for a 4-way set associative TLB
Therefore, the number of index bits = 2 (as there would be 4 sets in each way)
Size of virtual page number = 24 – 12 -2 = 10 bits
Therefore each TLB entry would contain the following -
Used bit---Dirty Bit---Valid Bit---Virtua page number (tag)---Physical page number (data)
----1----------1--------------1---------------10----------------------------------4----------
total of 17 bits

5) Size of cache = 256 Bytes
2-way set associative (LRU replacement)
16 byte block size
16-bit physical address
Number of sets in each way of the cache = (256B/(16B*2)) = 8 sets
Offset Bits = 4   Index Bits = 3   Tag bits = 9
Address sequence – 0x0110, 0x001C, 0x0018, 0x0010, 0x0484, 0x051C, 0x001C, 0x210, 0x051C
Hence, out of the 16-bit physical address top 9 bits would be used for tag comparison, next 3 bits would determine the set and the last 4 bits would give the offset.
Initial state -
Way 0 -

Set	Tag	Valid
0	X	0
1	X	0
2	X	0
3	X	0
4	X	0
5	X	0
6	X	0
7	X	0

Way 1 -

Set	Tag	Valid
0	X	0
1	X	0
2	X	0
3	X	0
4	X	0
5	X	0

6	X	0
7	X	0

On executing the above sequence the resulting cache state would be -
All the above addresses are targeted to the 1st set only except 0x0484 which is to set 0. The cache would encounter 3 hits and 6 misses. The final state would be -
Way 0 -

Set	Tag	Valid
0	0x9	1
1	0x4	1
2	X	0
3	X	0
4	X	0
5	X	0
6	X	0
7	X	0

Way 1 -

Set	Tag	Valid
0	0xA	1
1	X	0
2	X	0
3	X	0
4	X	0
5	X	0
6	X	0
7	X	0

Hit ratio would be = 3/9

b) Virtual address = 32 bits
Page size = 4KB
Physical address = 36 bits
L1 data cache -
Size = 32KB
Block size = 64B
2-way set associative
TLB -
4-way set associative
Total entries = 128
5 extra bits (dirty, reference, 3 permission bits)

L1 cache -
Physically tagged and indexed
Physical address bits = 36-bits

Block Size = 64B -> number of offset bits = 6-bits
Cache Size = 32KB
Number of sets in each way = (32KB/(64B * 2)) = 256 sets in each way
Number of index bits = 8-bits
Tag bits = 36 – 8 – 6 = 22-bits
Offset = 6, Index=8, Tag=22

TLB -
Will store virtual page number, physical page number + 5 extra bits
Total entries = 128
Entries in each way = 128/4 = 32 entries
Page size = 4KB
Offset bits = 12 bits
Since there are 32 entries, hence the index bits would be equal to 5-bits
Virtual adrdress = 32-bits
Virtual page number (tag bits) = 32 – 5- 12 = 15-bits
Offset = 12, Index=5, Tag=15

7) Three types of misses in the cache are –

i) Compulsory Miss – This type of cache miss occurs as the initial accesses won't be present in the cache. Hence, the data must be brought to the cache. Since, this type of miss would occur for all sizes of the cache, the only way to prevent this would to pre-load the cache with the data. This can be done by prefetching the data into the cache depending on the data access patterns.
ii) Capacity Miss – As the name suggests, once the cache is full any new access not present in the cache would lead to a miss. Since, the capacity of the cache is limited hence there would be capacity misses when the cache cannot contain all the data required by the program. One way to reduce such misses would be to have a larger cache such that more blocks could be brought into the cache.
iii) Conflict Miss – Conflict misses occur when there is a different tag access to the same index and the older data would needed to be removed. This can be reduced by increasing the associativity of the cache. It is true that a fully associative cache wouldn't result in this type of cache misses.

8) Cache optimization techniques for improving –

a) Miss rates – Miss rates in the cache can be categorized into misses due to cache being empty, misses due to cache being full and misses due to different accesses targeting same set. This leads to the cache miss and thus miss rates increases. In order to reduce the miss rate –
i) Bigger caches – On increasing the cache size the misses due to cache being full would reduce as the cache would have more sets and can save more data. This would definitely improve the miss rate for a large program.
ii) Higher associativity – By increasing the association in the cache we are basically creating more slots for the accesses to the same set. This would improve the hit rate by ensuring that there are less evictions during the accesses to the same set. This would reduce the conflict misses and hence the miss rate would reduce.
iii) Larger block size – Larger block size we would be exploiting the spatial locality and thus reducing the miss rate. Larger blocks would reduce the compulsory misses since more data would be stored per block hence taking advantage of the spatial locality.

b) Reduce miss penalty – Miss penalty is defined by the number of cycles/time it would take to replace the block of data from memory as a result of cache miss. This can be reduced by -
i) Multilevel Cache organization – The most effective way to reduce the miss penalty is to have a multilevel cache organization. Having a slightly slower but a bigger cache would reduce the miss penalty as the processor would still get the data quickly when compared to main memory.
ii) Priority to read misses over write misses – Since most of the processors today implement a write-buffer which store the write value could potentially lead to read-after-write hazard through memory if the write buffer isn't checked. On checking the write buffer, the data could be forwarded to the read request thus reducing the miss penalty provided there are no conflicts and the memory is available in the write buffer.
iii) Critical word first – This technique reduces miss penalty by sending the critical (required word) first before other data is in request. This would mean that the data would be made available a bit quickly rather than first copying the entire cache line and then offsetting into it. This would ensure less cycle penalty on misses.

c) Reduce Hit time – As the term suggest, hit time is the time it takes for certain request to hit in the cache and get the data out from it. It can be reduced by -
i) Small & simple level 1 caches – By having small caches it would take less amount of logic and thus the time to retrieve the data from the cache. This could further improve if we could possibly use a direct-mapped cache or to improve hit rate a 2-way set associative cache organization for a reduced hit time.
ii) Way prediction – Since most of the caches these days are usually n-way set associative, it would take some amount of time to compare data from every single way. The hit time can be reduced if we can have a way predictor which predicts the cache-way for a particular request. A good predictor would definitely reduce the amount of hit time taken by the cache.
iii) Virtually indexed and Physically tagged caches – As most of the processors contain virtual memories it is essential to avoid indexing the cache with physical index and physical tag comparison. One way to reduce hit time is to have virtually indexed caches. By the time the address is being translated the index information would be available (using the virtual address) and thus parallelising the cache lookup process. This would again reduce the hit time.

9)

a) Different ways in which compiler can reduce the cache misses in the program -
i) Loop interchange – By exchanging the nested loops such that the data is accessed in the order it is stored
ii) Blocking – By accessing the 2-D array in blocks instead of rows/columns thus taking advantage of temporal locality. Main idea is to have more cache hits by making use of the data already in the cache rather than replacing the data.
iii) Compiler hints – The loads/stores can get certain compiler hints on the cacheability of those instructions. If certain store data won't be required immediately, compiler can hint the hardware to not to allocate such data into the cache.
iv) Compiler controlled prefetching – In this way compiler can insert the prefetch instructions (should be supported by the ISA) to create interesting data patterns which reduce cache misses.

b) Different types of prefetching techniques are –
i) Hardware based prefetching – In this technique the hardware prefetches data/instructions into the respective caches. Usually on a cache miss the instruction and next instruction are prefetched. The current instruction goes into the cache and the next instruction is placed in the prefetched buffer. If the requested block is already present in the instruction stream buffer the cache request is cancelled and the next prefetch request is made. Similarly, this can be used for data accesses as well. By guessing the pattern in which a particular loop accesses data, the hardware prefetcher could prefetch the data well in advance in the caches to get a cache hit thus improving performance.
ii) Compiler controlled prefetching - In compiler controlled prefetching technique the compiler can insert the prefetch instructions (should be supported by the ISA) to create interesting data patterns which reduce cache misses. There are two types of prefetches possible –
1) Register prefetch – to load value into the register
2) Cache prefetch – to load value into the cache
The main goal is to overlap the instruction execution with the fetching of data. In case of the loops, the compiler can unroll the loop to have a better such that the different accesses would hit in the cache as the compiler schedules the optimization.

Prefetching can be bad if it prefetches addresses which might result into page faults or worse if it changes the processor state. Also, in case of hardware based prefetching the performance might reduce if the prefetcher cannot work in parallel with the execution. Such a prefetcher could possibly lead to stalls in the executing pipeline. If the prefetched data isn't being used by the program then it could lead to cache pollution (i.e cache containing data not required) and also can have impacts on the power.

c) Blocking in simple term means accessing data in blocks rather than using rows/columns thereby exploiting the advantages of temporal/spatial locality. The blocked algorithms work on blocks instead of rows or columns. The main idea is to store the data in the cache and then re-use that data as much as possible before moving to the next set of data. It is very essential for applications working with high dimensional array structures.

It reduces cache misses by exploiting temporal locality hence maximising the data access already loaded into the cache before moving to new sets of data which may lead to evictions. It tries to take advantages of both spatial and temporal locality thereby leading to reduced miss rate.

d) 1 MB L2 cache
Block size = 64B
Refill path size = 8B
Time to receive first 8Bytes from memory = 120 cycles
Additional 8 bytes = 16 cycles

L2 cache miss without critical word first -
The time to get the data from the cache in the case where the data requested belongs to the last work of the 64B block -
Time taken = 120 cycles + 7 x 16 cycles = 232 cycles (as the data would be present in the last 8B)

L2 cache miss with critical word first -
The time to get the data from the cache in the case where the data requested belongs to the last work of the 64B block -
Time taken = 120 cycles (since the data would be present in the first 8B only)

e) Non-blocking caches – Non-blocking caches makes sure that the cache keeps on providing hits for next instruction/data accesses when there is an on-going miss from the cache. This kind of technique is known as hit-under-miss i.e. the cache allows the next request to get a hit even when there is an on-going cache miss.
The next-generation modern processors would typically be out-of-order which means that they wouldn't need to stall on any kind of dependency and/or cache/TLB misses. In such a case a non-blocking cache would definitely help the processor in executing more instructions/data accesses as it would be able to give hit information for the next requests as well.

10) Processor clock rate = 1GHz = 1 ns clock cycle
Main memory access time = 100ns
L3 cache access = 20ns
L2 cache access = 10ns
L1 cache access = 1ns

L1 miss rate = 10%
L2 miss rate = 5%
L3 miss rate = 1%

a) Miss penalty for main memory = 100 ns/1ns = 100 cycles
Similarly, for L3 it would be = 20 cycles, L2 = 10 cycles

b) AMAT if there is only L1-cache in the system -
AMAT = Hit time + (Miss rate x Miss penalty)
Hit time for L1 cache only = 1ns
Miss rate = 0.1
Miss penalty = 100ns (since there is no L2/L3)
AMAT = 1 + (0.1*100) = 11 ns

c) On adding L2 to the system -
AMAT = Hit timeL1 + Miss rateL1 * (Hit timeL2 + (Miss RateL2 * Miss penaltyL2)
AMAT = 1 + (0.1) * (10 + (0.05*100)) = 2.5 ns
%Improvement = 11 – 2.5 / 11 * 100% = 77.27%

d) On adding L3 to the system -
AMAT = Hit timeL1 + Miss rateL1 * (Hit timeL2 + (Miss RateL2 *(Hit timeL3 + Miss rateL1* Miss penaltyL3))
AMAT = 1 + (0.1) * (10 + (0.05*(20 + .01*100)) = 2.105 ns
%Improvement = 2.5 – 2.105 / 2.5 * 100% = 15.8%

11) Direct-mapped cache
Block size = 64B, Total of 256 blocks
Offset bits = 6, Index bits = 8, Tag bits = 18
Size of float = 4B -> Each block would contain => 16 data samples
Arrays stored in row major order. Collision when the following pair is accessed a[i][j] and a[i+4][j]
Original code -
In case of the original code every 4th access by the inner loop would lead to a cache miss with the cache evicting the data. On every 4th access the set would rotate since the address would cross the 256 sets and come back to the first set. On a single complete iteration of the inner loop, the cache would contain the following contents –

Block	(Addr)	Valid
1	a[1020][0]	1
2	a[1021] [0]	1
3	a[1022] [0]	1
4	a[1023] [0]	1
..	..	
256	x (junk)	

This would lead to 1024 misses

The next iteration of the loop would be

Block	(Addr)	Valid
1	a[1020][1]	1
2	a[1021] [1]	1
3	a[1022] [1]	1
4	a[1023] [1]	1
..	..	
256	x (junk)	

This would again lead to 1024 misses.
Hence, all the accesses would lead into a miss (1024*1024 misses). Hit rate = 0%

Compiler Code –
In this case the first access would lead into a miss and then there would be 15 hits since those accesses would corresponds to the same cache lin. Hence, on one complete execution of the inner loop we would have 1024/16 = 64 misses
The cache would repeat on the 4th access of the outer loop. Thus there would be a total of 4*64 = 256 misses on every 4*1024 accesses. Total number of misses = 256*256.
Thus, hit rate would be = 256/(4*1024) = 1/16

12)
a) DIMM – DIMM stands for dual inline memory modules. These modules usually contain 4-16 DRAMS and are organized as 8-bytes wide. Due to this the associativity increases as each package would contain 4-16 DRAMS and thus leading to an improved performance.
Rank – Since many DRAMs are present in a single DIMM, we need some way to access them. The rank is used as the chip-select for the set of rams connected together and hence can be accessed simultaneously. This therefore improves the access time as the hardware would be able to access

multiple DRAMs simultaneously.
Bank – To improve performance, it would be better to access the RAMs not as a single monolithic block but it can be divided into multiple banks. In this way a single access can lookup into all the blocks simultaneously thereby improving performance as well as the access time.
Row buffer – On reading data from the cache it would be better to have the entire row of data accessed rather than accessing the required bytes. The row buffers thus reads a lot of data and sends out based on the addresses required.
b) SRAM – SRAM stands for static random access memory. Each SRAM cell usually consists of 6 transistors. The accesses from the memory are fast as compared to DRAMs but are quite costly. Due to this SRAMs are typically used for caches since they are smaller in size.
DRAM – DRAM stands for Dynamic Random access memory. The storage cell in a DRAM is a capacitor. Because of the leakage in capacitor, DRAM needs to refreshed every few milli-seconds to retain the data in the DRAM. These memories are cheap as compared to SRAM and most of the main memories are DRAM based.
Flash – The trends in energy and size of requirements ultimately lead to a new type of memories – Flash memories. This is a non-volatile memory and a standard mode of storage these days. These are the most cheap memories (even 15-20% cheaper than DRAM) but are slow when compared to the other two. These are usually build from a technique called EEPROM. Unlike DRAMs which are dynamic these memories are static in nature but they have a limited number of write cycles per block since these need to be erased before over-writing them.

c) Difference between the access time and cycle time of DRAM is that the access time is basically the time spent between the read request comes and when the data arrives on the other hand cycle time is the minimum time between certain unrelated events in the memory.
Since the storage element in the DRAM is a capacitor, DRAM suffers a basic problem of leakage. As capacitor cannot hold its charge after certain amount of time, in-order to avoid data loss, the DRAM needs to be refreshed every few milli-seconds.

d) The main difference between DRAM and DDR-DRAM is the rate at which the RAM responds. DRAM can only be accessed or basically can transfer data on either the positive edge or the negative edge of the clock. On the other hand DDR-DRAM can transfer data on both the positive and negative edge of the clock thereby increasing the bandwidth of data accesses.
Different standards available for DDR-DRAM are -
DDR, DDR2, DDR3, DDR4 – the main difference is in the DRAM performance when compared on clock rate, M transfers per second. The clock rate would twice of the clock rate for DDR for DDR2 and thrice in case of DDR3. The transfers per second would also follow the same numerical relationship.

e) Virtual Memory – It is a concept in which the physical memory is divided into blocks and helps many different processes to work simultaneously by allocating different blocks to each process. It can also help in providing security as the different pages of the physical memory can be mapped for either read only or write only thus preventing faulty accesses to that page. In order to support the virtual memory, there should be an address translation system available in the hardware and to improve performance a translation look-aside buffer (TLB) should be used as well.
Virtual Machine – Like the name suggests a virtual machine is an emulation of a physical computer system and provide functions which a physical compute can provide. This requires specialized

hardware as well as software. The first and foremost requirement is the need of the Instruction Set architecture to support VMM. Without the support it would not be beneficial to implement virtual machine. The hardware requirements are similar to virtual memory as the VMM would need a TLB and a translation regime. VMM also requires sharing of almost all the hardware resources thus this should be supported by the software (OS).

13) The author initially mentions that there is a growing focus towards accelerators and the feature-rich memory systems. The later part i.e. the feature-rich memory systems is the focus of the paper and the paper highlights the works of different authors on this subject. The author believes that the new era would see more benefits in features and it would be best if there could be certain features which could be added to the memory systems as well.
The author has defined the two main memory system features – processing features which provide certain logic to execute parts of an application within the memory system only and the other are auxiliary features which are independent of the application but would be useful for the overall system. These two features are discussed in more length in the paper. Some of the highlights of these features are –

i) Processing features – The idea to have certain processing element sharing the memory systems die is to exploit the benefits given by higher bandwidths. This processing technique is referred as the "Near Data Processing" (NDP) throughout the paper. The main motive behind near data processing is to reduce the memory bandwidth overhead and the energy spent in data movement for certain trivial data searching functions. In the case of NDP there would be very less latency thus leading to less energy dissipation over time. This form of processing is further enhanced if the running application could be parallelized. If the computation can be parallelized across the cores present in the memory devices, the benefits would be even more. The author describes how an in-memory MapReduce application which exhibits a very high degree of locality and parallelism can be benefitted in terms of efficiency and speed using these NDPs. Though with the NDPs enjoying the benefits of the high bandwidth, there are few issues also associated with this approach. One of such issue is how is virtual memory handled. The author says that it should be best done within the processor and the NDP should only touch the cache-line of interest but then it would again add a bit of delay since the processor would now have to forward all the required information (like physical address, cache line, etc) to the NDP. Another issue pointed out in the paper was related to the programming techniques. It was shown in the paper that the NDP would yield higher results in terms of efficiency and time if the program was parallelized. This adds the programmer with the burden of not only parallelizing the program but to decide the computation place of the program as well. Though a sophisticated hardware can manage this by selectively sending out certain computations to NDPs. These processing elements could be present as a localized core or as a 3D stacking near the DRAMs. The later approach proves to be more efficient but the author warns that such a technology won't be cheap and there is still a lot of on-going research in this area.

ii) Auxiliary features – As stated above these are the features which are totally independent of the running program but are still necessary for the overall efficiency of the system. Auxiliary functions may include functionalities like compression, memory timing, error-correction code, security. Since all these features are independent of the running application, the author believes that it would be best to have these features placed in-memory such these could be benefited with higher bandwidths. Also, certain techniques responsible for adding reliability to the system like the errorcorrecting codes are handled by the processor. If such a logic could be moved outside the processor it would reduce this overhead from the processor and might lead to a better performance. Certain security related functions could also be added to the DRAM memory system itself, though the author warns about attackers which could possibly take advantage if plain text is sent out of the processor. Thus, the processor would still need to send an encrypted version of the data but there can be techniques which could offload some of this load from the processor. Another important feature is related to the timing of reads/writes to the memory. Due to process variations different memory regions might behave differently in-terms of faster response and some parts could be more prone to errors. The author believes that this worst-case timing parameters computation could be offloaded from the processor and moved to memory which can offer some low-level timing management.

The need for all these new features is to have sophisticated as well as specialized memory systems. This is relatively a new area of research and thus can lead into many new innovative products. Since the processors have now almost reached the peaks of innovation, in order to have more specialized systems it would be necessary to look into different areas. The memory system continues to be the bottleneck of the modern day processor, thus it would be best to have some innovative techniques to reduce the delay associated with the memory system. Using the above two mentioned features, the author wants to create a new specialized memory system which would yield in better performance and more innovative solutions!