Arquitecturas de Alto Desempenho

Memory Hierarchy Design

António Rui Borges

Summary

- *Why memory management is so important*
  - *Principle of locality*
  - *Memory hierarchy*
- *Cache*
  - *Cache principles*
  - *Cache performance*
  - *Cache optimization*
- *Main memory*
  - *Asynchronous dynamic RAMs*
  - *Synchronous dynamic RAMs*
- *Suggested reading*

Why memory management is so important - 1

Since the early days of the computing era, programmers dream about *unlimited* amounts of *fast* memory to store and run their programs, that is, they dream about having as much memory as they deem necessary to store code and data which may then be accessed at the maximum rate the processor can operate.

Although *main memory* capacity has increased steadily over the years, the fact is programs tend to expand in size and complexity filling all the space that is available – *Parkinson Law*.

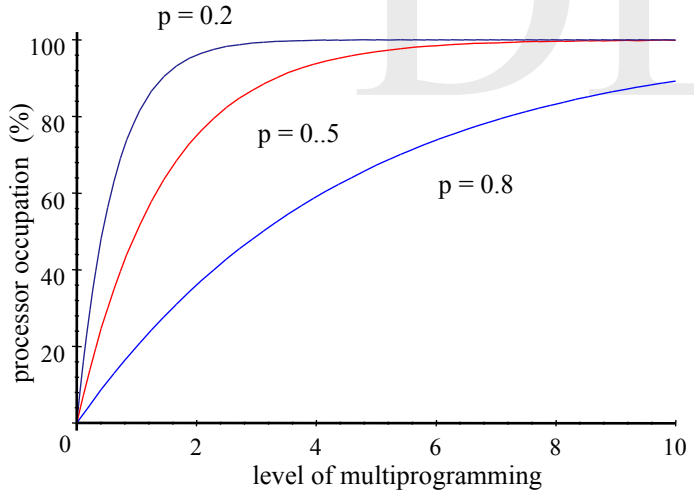
This is true not only because programs have become more complex as the performance of computer systems improves, dealing with problems that could not have been previously tackled, but also because it is critical in a multitasking environment to store together in main memory the addressing spaces of many processes so that the processor is kept fully occupied and the associated *response* and/or *turn-around* times are minimized.

Why memory management is so important - 2

fraction of processor occupation $= 1 - p^n$ (simplified model)

p - fraction of the time a process waits blocked for the completeness of I/O operations, synchronization or any other cause

n - number of processes which currently coexist in main memory



N. of processes in MP	*% of occupation (P)*
4	59
8	83
12	93
16	97

$p = 0,8$

Principle of locality - 1

It is not possible, however, to satisfy literally programmers' dream.

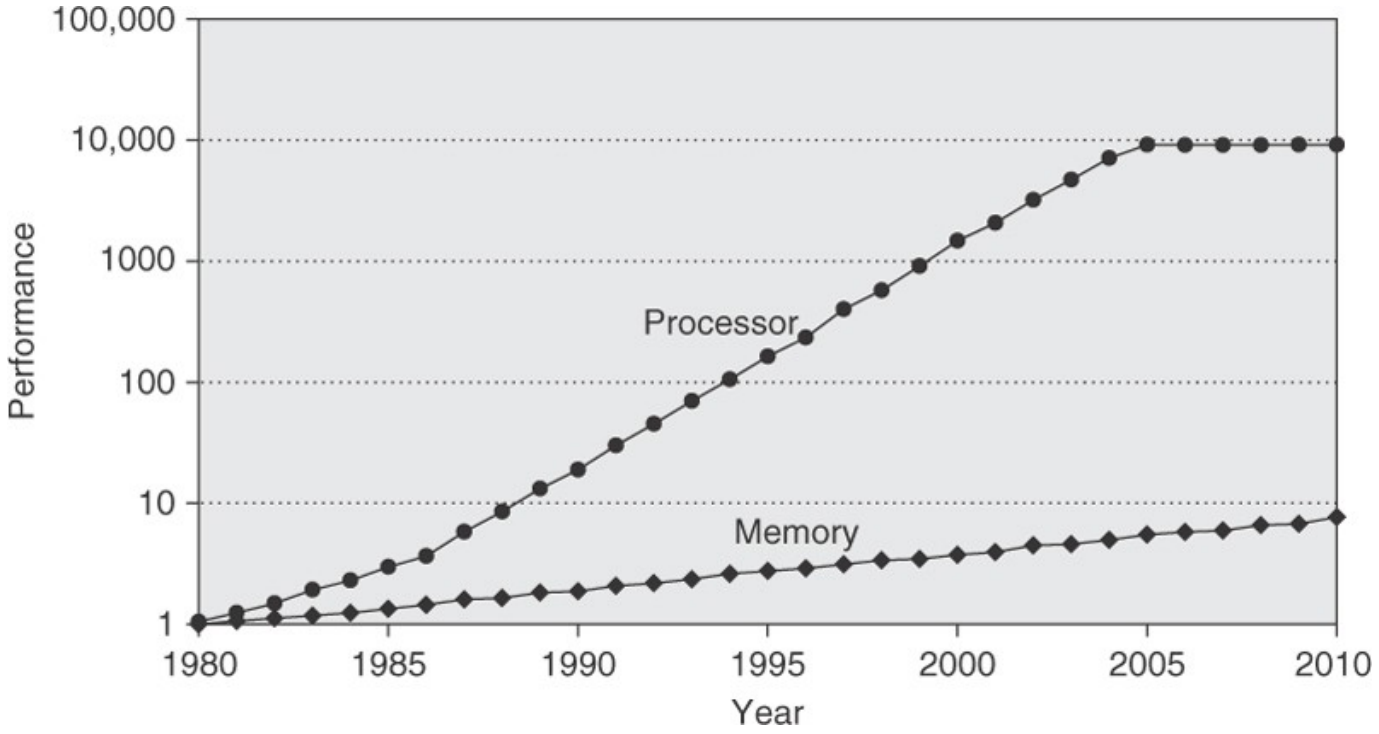
It can be shown that, for a given implementation technology and a given power budget, simpler hardware can be made faster. As complexity grows, signal propagation time delays become longer due to the increase of the required amount of interconnection circuitry and the decrease of the intensity of the driving electrical currents.

Futhermore, the problem has turned more severe as time went by. Instead of being reduced, the processor-DRAM performance gap has augmented gradually over the past decades and has even worsened since the introduction of multicore processors, where the aggregate peak bandwidth is proportional to the number of processors in the core.

Principle of locality - 2

Over time performance variation of single processor vs. memory

Source: Computer Architecture: A Quantitative Approach



processor curve – average number of memory requests per second
memory curve – inverse of DRAM access latency

Principle of locality - 3

To deal with this problem, the approach followed by memory systems designers is based on an observational fact derived from the tracing of program execution and known as the *principle of locality*. It simply states that, for relatively large periods of time, a program tends to reference a very definite fraction of its addressing space. Thus, one may speak of

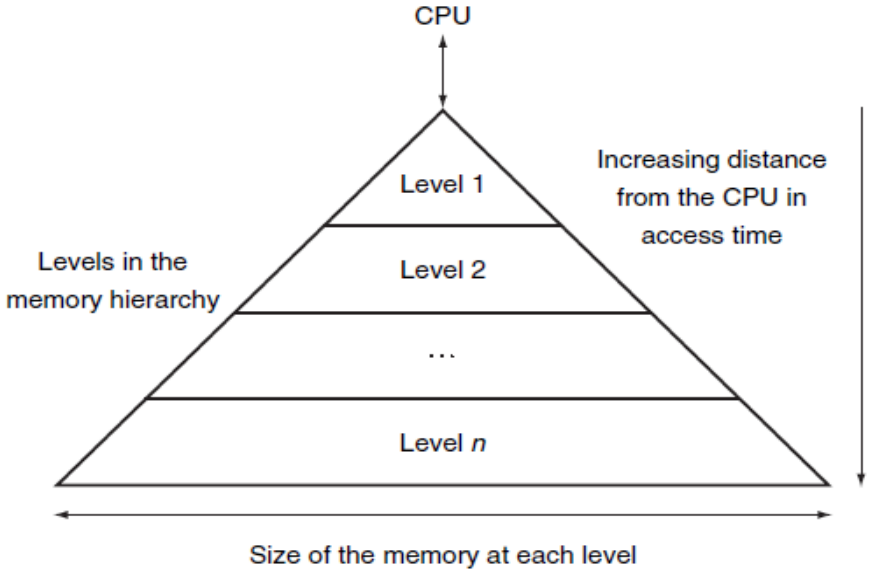
- *spatial locality* – when a memory word is referenced once, another one, which is stored nearby, may be referenced soon
- *temporal locality* – when a memory word is referenced once, it may be referenced again soon.

It is not too difficult to see why programs have this kind of behavior. Try to assert the reasons why it is so!

Memory hierarchy - 1

To take advantage of the principle of locality, computer memory is implemented as a memory *hierarchy*, that is, it consists of multiple levels of memory with different sizes and access speeds. As a rule, the faster memory modules have a higher price per bit than the slower ones and a smaller storage capacity.

The goal is to present the programmer with as much memory as it is available in the cheapest technology (lowest level), while providing access at the speed offered by the most expensive one (highest level).



Source: Computer Organization and Design: The Hardware/Software Interface

Memory hierarchy - 2

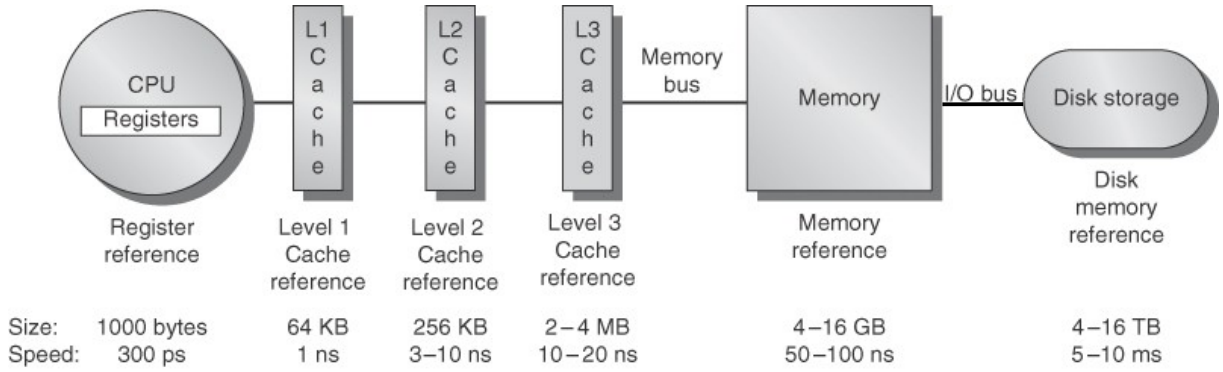
Memory hierarchy may be divided into distinct levels

- *register bank* – internal memory to the processor, access is controlled by the instructions
- *cache* – external memory to the processor, access is controlled by *instruction fetch* and *data transfer* instructions; it presently consists of several levels of static RAM, all of them placed inside the processor integrated circuit; the first level is even located inside the processor chip and is split into instruction and data units
- *main memory* – external memory to the processor; it implements the *stored-program concept* defining the device where instructions and data of an executing program are mostly stored; it consists of dynamic RAM
- *swapping area* – non-volatile memory located in mass storage; it works as an extension of the main memory to implement an operating system controlled memory organization, usually a *virtual memory paged-architecture*; it consists mostly of HDD or flash memory devices.

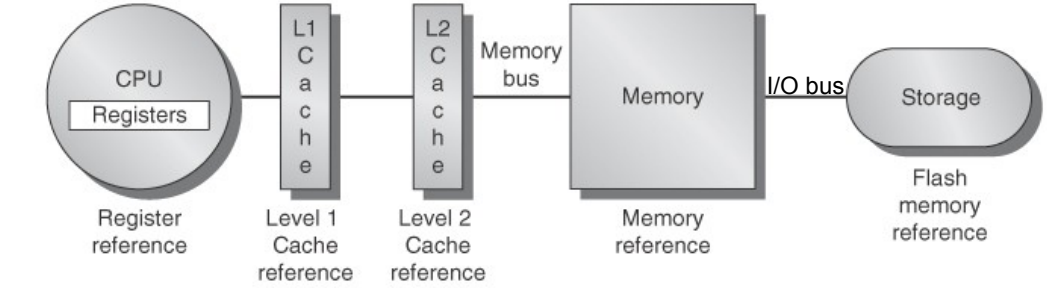
Memory hierarchy - 3

Levels of a typical memory hierarchy

Source: Computer Architecture: A Quantitative Approach



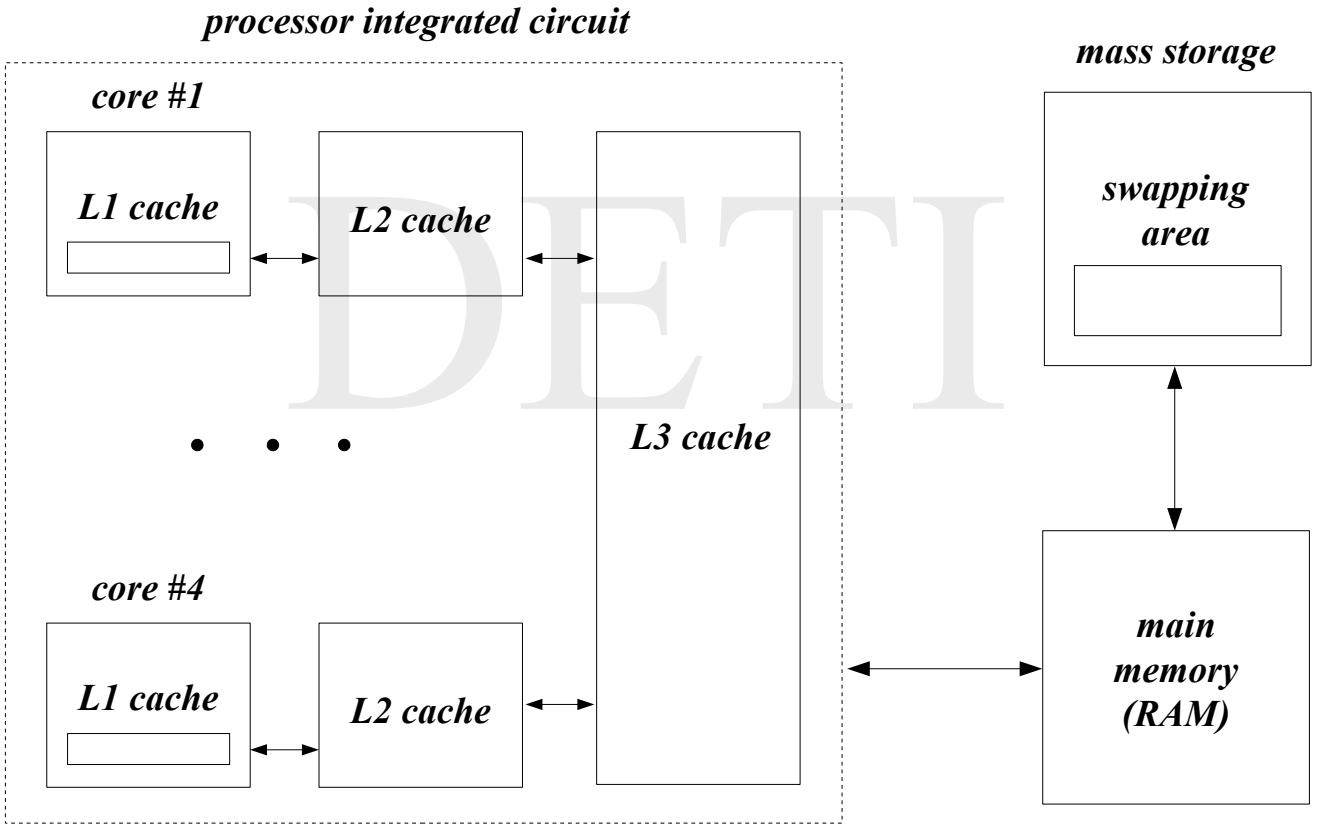
(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

Memory hierarchy - 4

Typical memory hierarchy for a 4-core processor system



Memory hierarchy - 5

Technical specifications of a typical memory hierarchy

Source: adapted from Computer Architecture: A Quantitative Approach

Name	*Typical size*	*Implementation technology*	*Access time (ns)*	*Bandwidth (MB/s)*	*Managed by*	*Backed by*
register bank	less 1KB	multiport custom design – CMOS	0,15 – 0,30	$10^5 – 10^6$	compiler	cache
cache	32 KB – 8 MB	on/off-chip CMOS SRAM	0,5 – 15	$10^4 – 4 \times 10^4$	hardware	main memory
main memory	less 512 KB	CMOS DRAM	30 – 200	$5 \times 10^3 – 2 \times 10^4$	operating system	swapping area
swapping area	greater 1 TB	semiconductor / magnetic	$3 \times 10^4 - 5 \times 10^6$	50 – 500	operating system	long term storage

Memory hierarchy - 6

In a memory hierarchy, the higher a level is, the closer to the processor it is located. Memory hierarchies take advantage of *temporal locality*, by keeping more recently accessed instructions and data closer to the processor, and of *spatial locality*, by moving blocks consisting of multiple contiguous memory words to higher levels of the hierarchy.

In most systems, memory constitutes a true hierarchy, that is, in order for data to be present at level *i*, it must be present first at level *i*+1, and all data is present at the lowest level.

The underlying concepts to building memory systems affect many other aspects beyond computer architecture. These include how the operating system manages memory and I/O, how compilers generate code and even how applications use the computer resources.

Because all programs spend much of their time accessing memory, the memory subsystem is a determinant factor for performance. Therefore, programmers need nowadays to understand that memory is hierarchic to improve the performance of their programs.

Memory hierarchy - 7

Although a memory hierarchy may consist of several levels, data transfer usually takes place only between two adjacent levels at a time, so one can concentrate on what happens between any pair of levels to get a general view of operations.

The minimum amount of data that can either be present or missing in a two-level hierarchy is called a *block*. We say that a *hit* occurs when the data requested by the processor appears in some block at the upper level; otherwise, the request is called a *miss* and the lower level is then accessed to retrieve the block containing the requested data.

The *hit rate* or *hit ratio* is the fraction of memory references to data found in the upper level over all memory references. On the other hand, the *miss rate* or *miss ratio* is its complement to 1.

Since performance is the major issue, the time required to service hits and misses is relevant. The *hit time*, being the time to access the upper level, comprises also the time to assert whether the access is a hit or a miss. The *miss penalty*, then, is the time to replace a block at the upper level by the block containing the data requested by the processor.

Cache principles - 1

Cache is the name traditionally given to the level(s) of memory hierarchy located between the processor and the main memory. Nowadays, however, the term has a wider meaning: it refers to any storage device which is managed in a manner that takes advantage of the principle of locality.

When dealing with the cache, the term *line* is specifically used to refer to the minimum amount of information that is transferred between any pair of cache levels or stored at the lowest cache level. *Block* is reserved to refer to the data itself either stored in a cache line or in main memory.

Assuming a single level cache, the answer to the following questions will help to enlighten the way how a cache works

- where is a line located in the cache? (*line placement*)
- how is it found if it were present there? (*line identification*)
- which line should be changed on a miss? (*line replacement*)
- what happens in a write operation? (*write strategy*).

Cache principles - 2

Since main memory capacity is much larger than cache size, many memory blocks will overlap at the same location within the cache over time. There are several ways for doing this, but one should bear in mind that the major goals are to keep hardware simple and the whole storing/accessing procedure efficient.

In this sense, the block size should not be completely arbitrary. It is important that the number of stored bytes be a power of 2 so that a *memory address* may be trivially split into a *block address* and an *offset*.

In general, cache can be organized as

- *direct mapped* – when there is a single place where a *block* may be placed
- *fully associative* – when a *block* can be placed anywhere
- *set associative* – when there are an aggregate of places, called a *set*, where a *block* may be placed.

Direct mapped to fully associative organizations form a continuum of levels of set associativity: the set size for *direct mapping* is one and for *fully associativity* is equal to the number of lines in the cache.

Cache principles - 3

memory address

block address (s bits)	offset (w bits)

Main memory / cache characterization

address length $= s + w$ bits

number of addressable units in main memory $= 2^{s+w}$ bytes

number of blocks in main memory $= 2^{s}$

number of lines in the cache $= m = 2^{r}$

cache size $= 2^{r+w}$ bytes

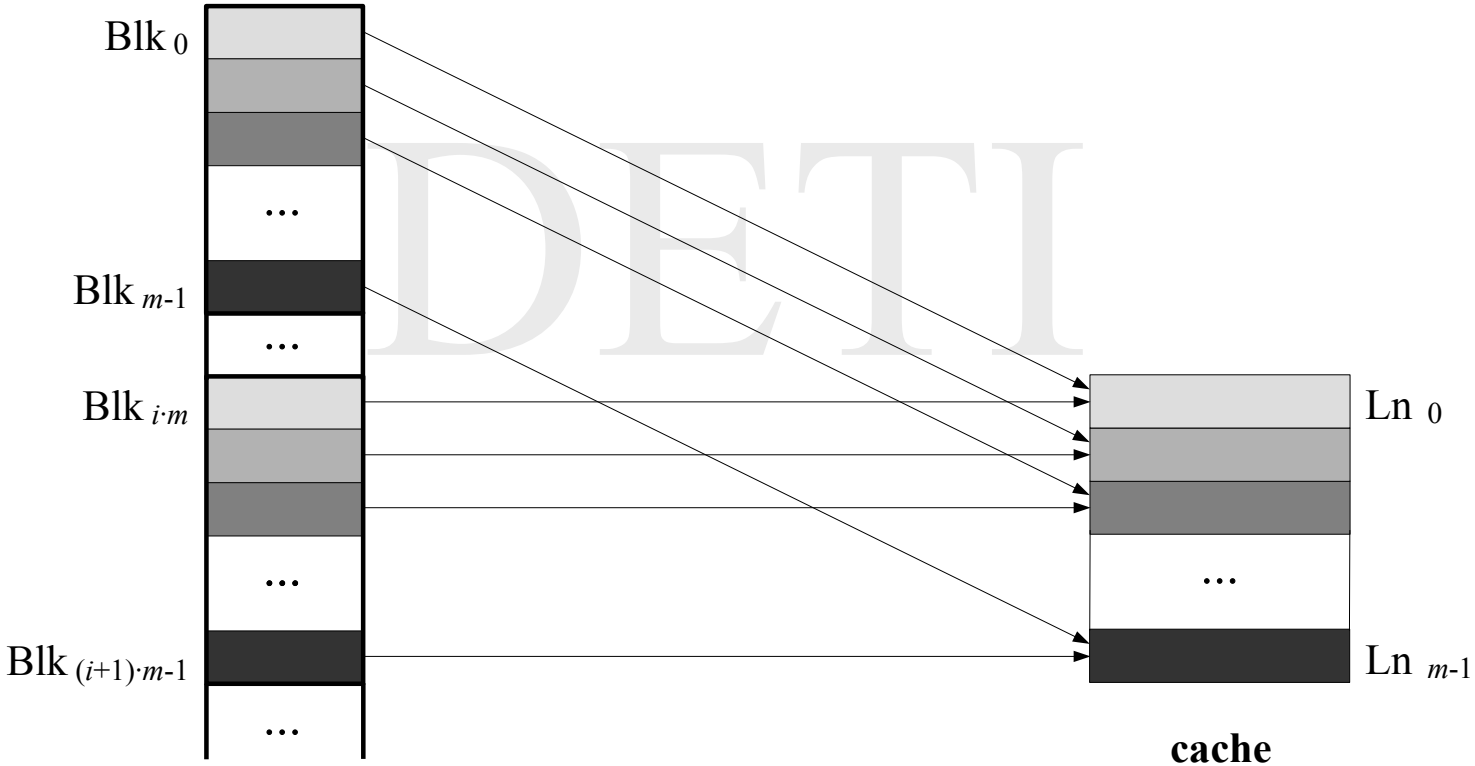
number of sets in the cache $= v = 2^{u}$

number of lines per set $= k = m / v = 2^{r-u}$

Cache principles - 4

Direct mapping

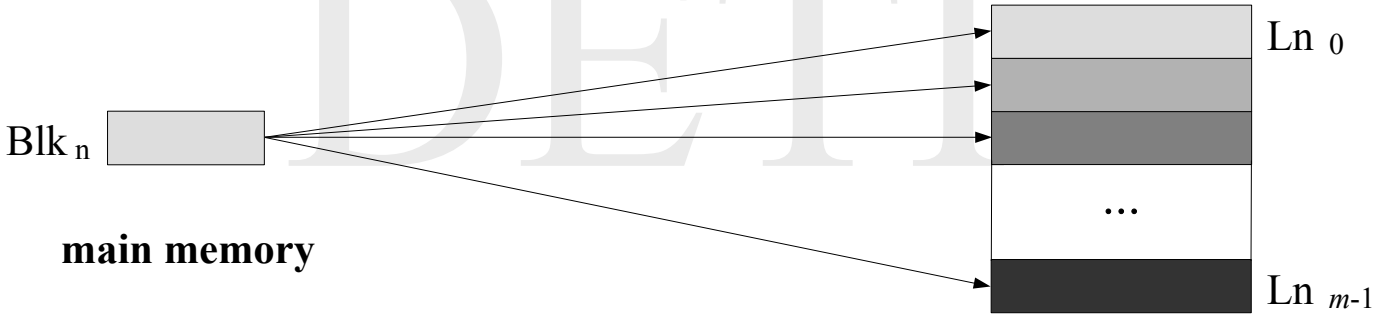
cache line address = block address *mod* number of lines in the cache



Cache principles - 5

Fully associativity

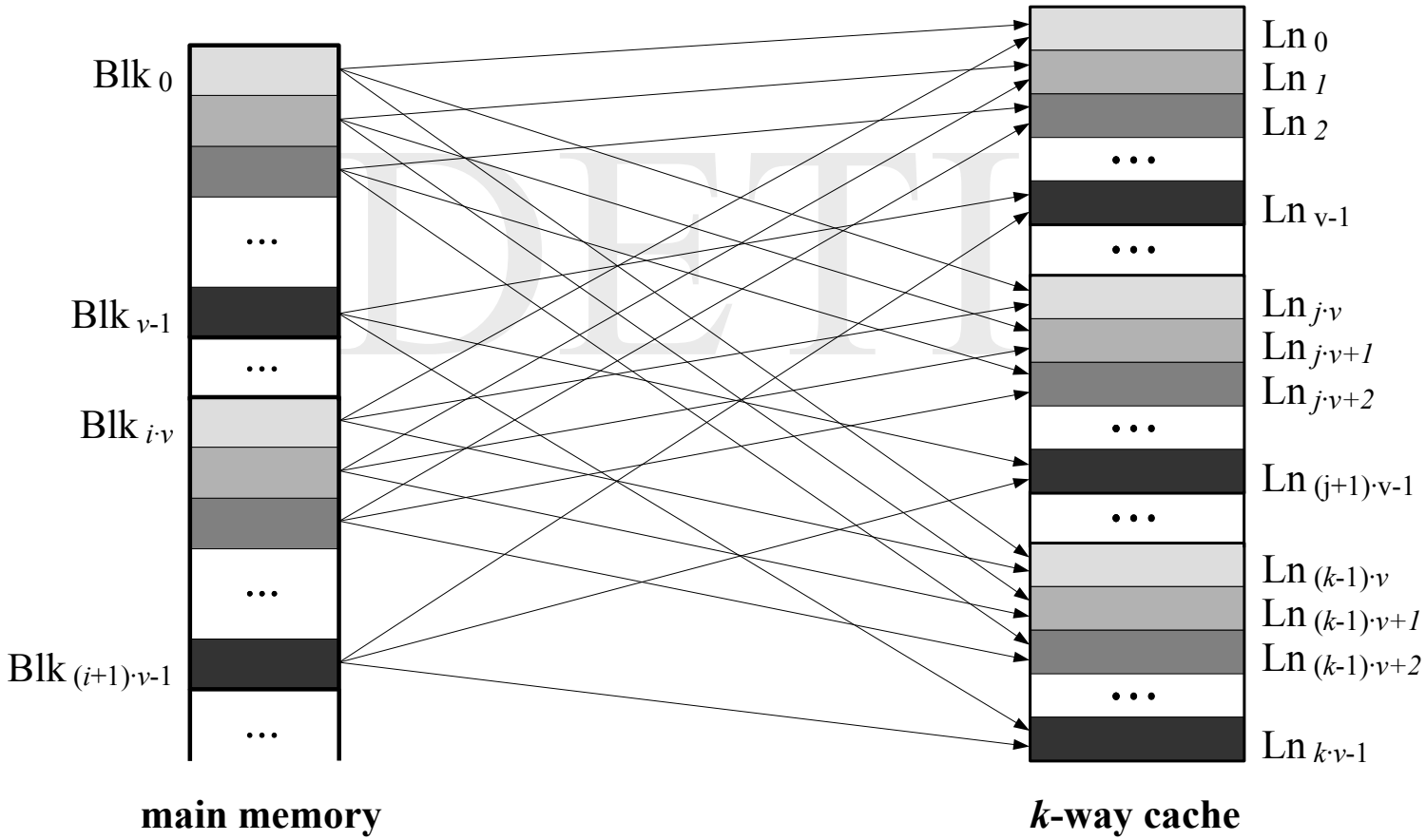
cache line address = any



Cache principles - 6

Set associativity

cache line address = block address *mod* number of sets in the cache



Cache principles - 7

Besides the contents of a main memory block, a cache line must also contain at least part of its address, usually called the block address *tag field*. The *tag field* is stored in every line of the cache so that it can be checked against the corresponding part of the memory address generated by the processor to determine if there is a match when an access occurs. As a rule, all possible *tags* are checked in parallel to make the search fast. Furthermore, since a data register is never empty, an extra bit (called the *validation bit*) is required to assert if the data presently stored in the line is meaningful or not.

Whenever the *tag field* of a block address is not the whole block address, the remaining bits of the address form a second field, the *index field*, whose goal is to select a specific set within the cache.

block address

tag field (*s*-*u* bits)	index field (*u* bits)

Cache principles - 8

Direct mapping

block address

tag field (*s*-*r* bits)	index field (*r* bits)

When the cache is organized through *direct mapping*, there is a single line within the cache where a particular memory block may be stored. This means that the number of lines per set is equal to one, the number of sets is equal to the number of lines and the tag field contained in each line has minimal length.

This organization leads to very simple, fast and efficient implementations. The main disadvantage is the risk of *thrashing:* a phenomenon that arises when the fraction of the addressing space referenced by the processor in an extended period of time contains groups of two or more addresses that map into the same cache lines. When this happens, the hit rate degrades quite a lot and program execution becomes rather slow because no benefit is taken from locality of reference.

Cache principles - 9

Fully associativity

block address

tag field (*s* bits)

When the cache is organized through *fully associativity*, all the lines within the cache are available for the storage of a particular memory block. This means that the number of lines per set is equal to the number of lines in the cache, the number of sets is equal to one and the tag field contained in each line has maximal length (the index field does not exist).

This organization leads to the minimization of the miss rate since in principle a specific memory block may be stored in any of the lines of the cache. The main disadvantage is the design complexity that it entails and, because of that, for a given implementation technology and a given power budget, it limits the speed of taking a decision for a hit or a miss.

Cache principles - 10

Set associativity

block address

tag field (*s*-*u* bits)	index field (*u* bits)

When the cache is organized through *set associativity* (*k*-way cache), there are exactly *k* lines within the cache where a particular memory block may be stored. This means that the number of lines per set is equal to *k* and the number of sets is equal to *v*.

This organization tries to attain *the best of the world* as portrayed by the two previous organizations. It leads to not too complex implementations that are still fast and efficient and avoids the risk of *trashing* by providing some redundancy to where a memory block may be stored.

Cache principles - 11

When a miss occurs, the cache controller must select a line whose block will be replaced with the desired data. With *direct mapped* organization, the problem is trivial since only one line is checked for a hit and only this line contents can be modified. With fully associative and set associative organizations, there are in principle many, or at least some, lines to be considered. If the *validation bit* for any of these lines is reset, one of them may be selected, but after some time all of them will contain valid data and so a decision must be taken.

The obvious strategy, the one that minimizes the miss rate, is to choose the line within the group whose data will not be referenced anymore or, if it will be, the reference will happen at the farthest distance from the present – the *principle of optimality*. Unfortunately, this rule is not causal, it would require guessing the future and, therefore, can not be implemented in practice.

Cache principles - 12

The main strategies employed for line selection are

- *random* – a pseudo-random generator is used to spread replacement uniformly among candidate lines; this potentiates a reproducible behavior
- *least recently used* (LRU) – in order to approximate the principle of optimality one relies on the past to predict the future; thus, the candidate line is the one which has not been referenced for the longest period of time
- *first in, first out* (FIFO) – because LRU leads to a complex implementation, an approximation to it that is simpler, but still relies on the past to predict the future, is to consider the line whose contents has remained for the longest period of time in the cache.

Cache principles - 13

Data cache misses per 1000 instructions for the Alpha architecture (DEC) using 10 SPEC2000 benchmarks (5 SPECint2000 and 5 SPECfp2000)

Source: Computer Architecture: A Quantitative Approach

Set associativity (block size = 64 bytes)									
	2-way			**4-way**			**8-way**		
Cache size	**Random**	**LRU**	**FIFO**	**Random**	**LRU**	**FIFO**	**Random**	**LRU**	**FIFO**
16 KB	117,3	114,1	115,5	115,1	111,7	113,3	111,8	109,0	110,4
64 KB	104,3	103,4	103,9	102,3	102,4	103,1	100,5	99,7	100,3
256 KB	92,1	92,2	92,5	92,1	92,1	92,5	92,1	92,1	92,5

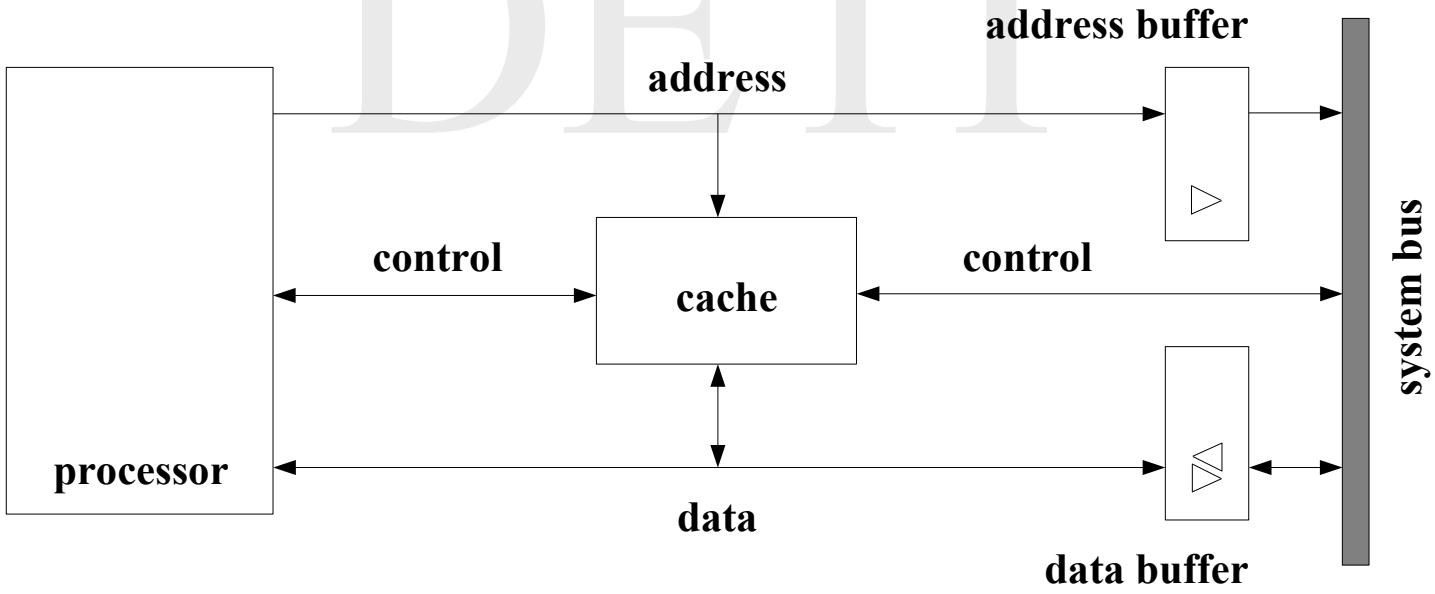
Cache principles - 14

Read operations from memory dominate processor cache accesses. This is so because all *instruction fetches* are read operations and most instructions do not write to memory. In fact, 10 SPEC2000 benchmarks (5 SPECint2000 / 5 SPECfp2000), running on a MIPS processor, suggest an average of 26% / 30% of *load* instructions and of 10% / 9% of *store* instructions over all executed instructions [Computer Architecture: A Quantitative Approach], which leads to a proportion of *read* over *write* operations of 93:7 / 94:6.

Making the *common case fast* means optimizing caches for reads, especially since traditionally processors wait for the completion of reads, but need not wait for the completion of writes. A *read* operation may start as soon as the block address containing the referenced data is made available. At the same time, the tags of candidate lines are compared to the *tag field* of the block address to determine if there is a hit. If so, the requested part of the block is immediately passed on to the processor; otherwise, the block transfer from the lower level is initiated and the read operation is stalled until the transfer is accomplished.

Cache principles - 15

In order to further optimize caches for read operations, contemporary organizations connect the pair processor-cache via address, data and control lines. Address and data lines also connect to address and data buffers that mediate the access to the system bus from which main memory is reached. When a hit occurs, address and data buffers are disabled and all the communication is internal. When a miss occurs, the block address is loaded onto the system bus and data are returned to both the cache and the processor.



Source: Computer Organization and Architecture Designing for Performance

Cache principles - 16

Write operations are a bit different. Modifying the block contents can only start after the tag is checked to determine if there is a hit. Thus, write operations take usually longer than the read operations. Besides, although the processor always specifies the data size and location, only for write operations this is critical because a definite portion of the block contents is changed; for read operations, the access to more data bytes than it is required, is irrelevant.

There are two basic write *policies*

- *write-through* – data are written to both the cache line and to the block in the lower level
- *write-back* – data are written to the cache line; the line block contents is only written to the lower level when the block is replaced.

When write-back policy is implemented, a key feature, known as the block *dirty bit*, is commonly employed to warrant that only modified blocks are transferred back on replacement. When a block is first transferred to the cache, its status bit assumes the value *clean* to signal that its contents has not been changed; when a write occurs, the status bit then assumes the value *dirty* and the block must be written back on replacement.

Cache principles - 17

Write-through advantages

- it is simpler to implement; since all write operations result in a write to the lower level, the cache lines are always clean
- the updated block contents is always present at the lower level, which simplifies the ensurance of data coherency
- it plays an important role in the design of multilevel caches; for the upper levels, the writes need only to propagate to the next lower level rather than all the way to the main memory.

Write-back advantages

- write operations for hits occur at the speed of the cache and multiple writes to the same block require only one write back to the lower level when the block is replaced
- less memory bandwidth is used, making it attractive for multiprocessors
- power is also saved, making it attractive for embedded applications.

Cache principles - 18

The processor stalls for the completion of write operations during a *write-through* procedure. A common optimization to reduce write stalls is to implement a *write buffer*, which allows the processor to continue as soon as data are written to the buffer, overlapping in fact processor execution with memory updating.

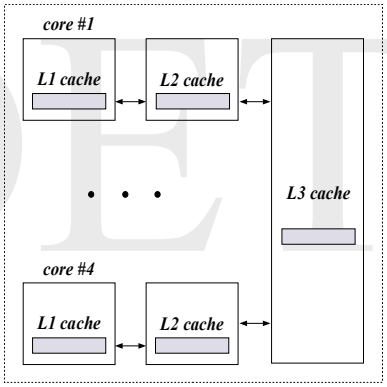
A *write miss* may be dealt with in the following ways

- *write allocate* – a line is allocated whenever a miss occurs, then the write operation takes place; however, if the block size is larger than data to be written, the block must first be retrieved from the lower level
- *non write allocate* – the cache is unaffected by misses, the write operation takes place only at the lower level, which means that blocks stay out of the cache until the processor reads data from them.

Either *write miss policy* can be used with any of the *write policies*. However, *write-back caches* usually implement the *write allocate* policy hoping that subsequent writes to that block are caught by the cache. In the same way, *write-through caches* implement the *non write allocate* policy because all writes must be written at the lower level, so no gain is obtained by the allocation of the block.

Cache principles - 19

For a multicore processor, where multiple simultaneous threads may be competing for access to shared data, protected or not by critical regions, a further problem arises which has to do with *cache coherence*. In such a situation, copies of the same memory block may be stored in lines of level 1 or level 2 caches associated with different processors.



In order to ensure that all processors always see the same data, a *write-through* policy should be implemented for level 1 and level 2 caches and when a *write* takes place, all copies at these levels should be made *stale* so that a transfer from level 3 cache is carried out if a subsequent *read* arises.

Cache principles - 20

During a program run, the processor not only fetches instructions, but also accesses memory to load and store operands. Although a single, *unified*, or *mixed* cache can supply both, it can generate a communication bottleneck. A typical situation where this happens is when a pipelined processor is executing a *load* or *store* instruction and, at the same time, is fetching the next one. In order to avoid this kind of structural hazard, it is common nowadays to have at the highest level two caches: one dedicated to instructions and another to data. Thus, the communication bandwidth with memory can be doubled through the use of separate ports.

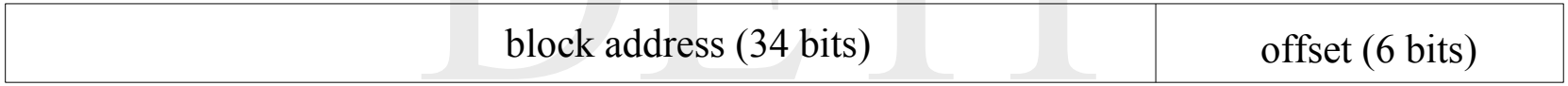
Dedicated caches also allow the individual tuning of each cache, by specifying different storage capacities, block sizes and associativities for each.

The Opteron data cache - 1

The Opteron data cache has a size of 64KB and is organized as a 2-way set associative cache, able to store data blocks of 64 bytes. It features a LRU replacement strategy and implements a policy of *write-back* with *write allocate* on write miss.

The Opteron presents a 40-bit memory address to the cache split as follows.

memory address



address length = 40 bits
number of addressable units in main memory = 2^{40} bytes / 2^{37} words
number of blocks in main memory = 2^{34}
block size = 2^{6} = 64 bytes

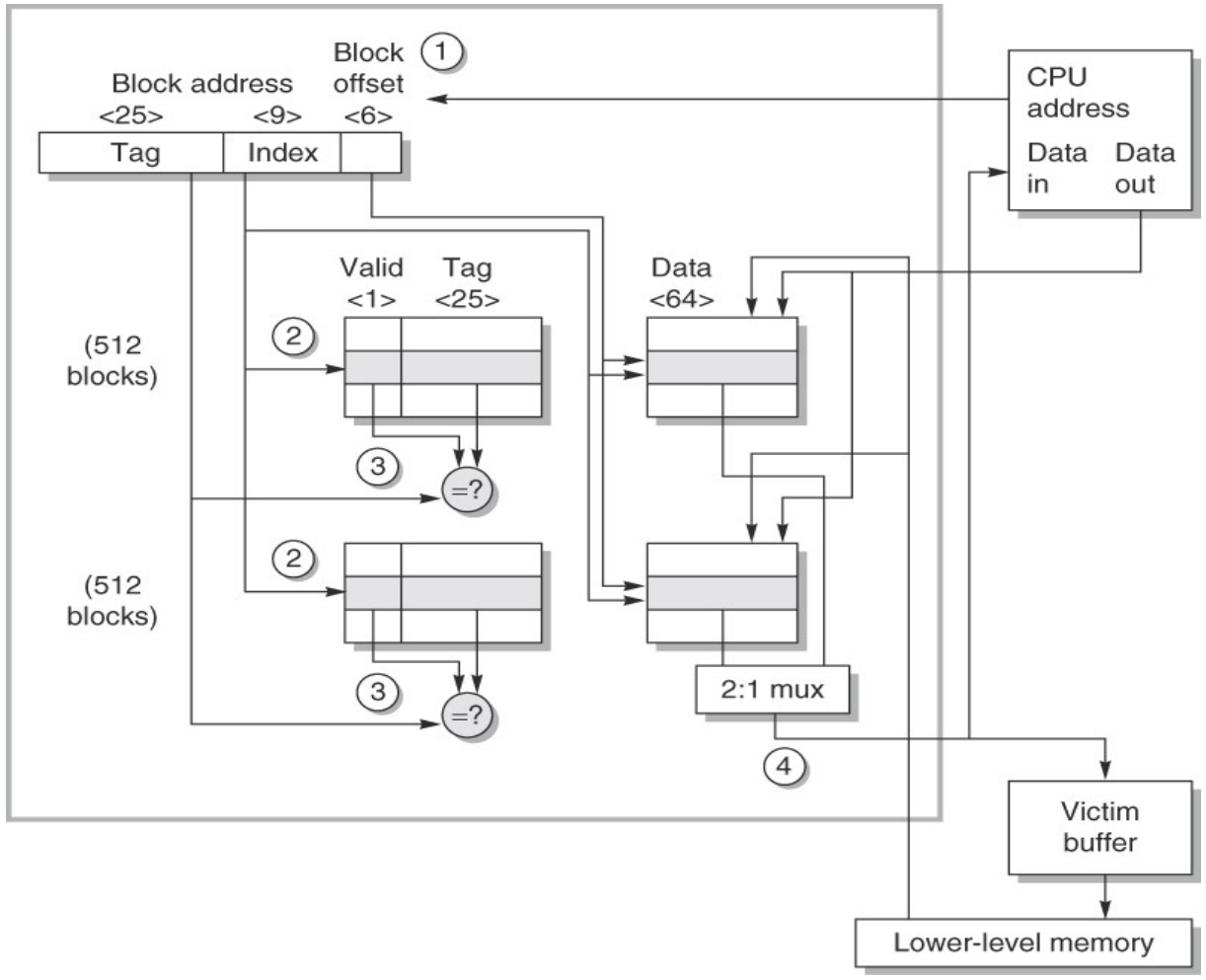
The Opteron data cache - 2

To compute the number of lines in the cache, we make

$$m = \frac{\text{cache size}}{\text{set associativity} \cdot \text{block size}} = \frac{2^{16}}{2 \cdot 2^{6}} = 2^{9} = 512$$

One point worth noting is that, since Opteron is a 64-bit processor, each memory access is at the most 8 bytes wide. Therefore, in addition to the 9 bits of the *index field* to select the proper memory block, the 3 most significant bits of the *block offset* are also used for selection of the proper word within the block.

The Opteron data cache - 3



Source: Computer Architecture: A Quantitative Approach

The Opteron data cache - 4

Several steps may be highlighted in an access to the cache

- *step* 1 – the memory address generated by the processor is split into two parts: the *block address*, the most significant 34 bits, which references a particular memory block, and the *block offset*, the least significant 6 bits, which references a specific byte within the block
- *step* 2 – the block address, in turn, is also split into two parts: the *tag field*, the most significant 25 bits, which serves to determine if the block is present in the cache, and the *index field*, the least significant 9 bits, which references the set within the cache where the block may be stored
- *step* 3 – if the *validation bits* are set, a comparison is carried out between the *tags* of the two lines that belong to the referenced set, and the *tag field* of the block address
- *step* 4 – when there is a hit, the cache signals the processor to proceed with the reading or writing of the data; however, since Opteron executes out of order, writing only occurs after the processor signals that the instruction has committed.

The Opteron allows 2 clock cycles for these four steps.

The Opteron data cache - 5

When there is a miss, the cache sends a signal to the processor informing that the data are not yet available. Next, the required block is read from the lower level of the hierarchy, since the cache implements a policy of *write-back* with *write allocate* on write miss. The latency is 7 clock cycles for the first 8 bytes of the block and, then, 2 clock cycles per 8 bytes for the rest of the block.

Due to 2-way set associativity featured by the cache, there are to begin with two lines where the block may be stored. If any of them does not contain valid data, its *validation bit* is reset, the line is immediately selected; otherwise, the *least recently used* rule is applied, that is, the selected line is the one whose *LRU bit* is reset. Notice that the implementation of the rule is in this case almost trivial: upon a hit, the line *LRU bit* is set and the *LRU bit* of the counterpart line in the set is reset.

Before doing that, however, because of the *write-back* policy, the *dirty bit* of the line whose block is to be replaced is checked. If it is dirty, its tag and data are first removed to the *victim buffer*. The cache provides space to store up to eight *victim* blocks which are later transferred to the lower level in parallel with other cache activities. If at some moment, however, the *victim buffer* is full, the processor has to stall until space is made available.

The Opteron data cache - 6

Format of a cache line

validation bit	**tag** (25 bits)	**dirty bit**	**LRU bit**	**data word** (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)
				data word (8 bytes)

Cache performance - 1

One method to evaluate *cache performance* is to expand the equation of *processor execution time*. The way to accomplish it is to elicit from the equation the number of clock cycles during which the processor is stalled waiting for a memory access, a parameter usually called *memory stall clock cycles*.

Then one gets

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{memory stall clock cycles}) \cdot \cdot \text{clock cycle time} .$$

Notice that the above equation assumes that the variable *CPU clock cycles* include the number of clock cycles to handle a cache hit and that the processor is completely stalled during a cache miss, the case of an *in-order execution* processor.

Cache performance - 2

The number of *memory stall clock cycles* depends both on the *number of misses* and on the *cost per miss* or *miss penalty*

$$\text{memory stall clock cycles} = \text{number of misses} \cdot \text{miss penalty clock cycles} =$$

$$= \text{instruction count} \cdot \frac{\text{misses}}{\text{instruction}} \cdot \text{miss penalty clock cycles} =$$

$$= \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot$$

$$\cdot \text{miss rate} \cdot \text{miss penalty clock cycles} \ .$$

Getting the value of *instruction count* is straightforward if one has a listing of the program compilation into assembly language. For speculative processors, however, only committed instructions should be counted. The value of *memory accesses per instruction* can be estimated in the same way: every instruction requires a memory access for instruction fetch and, according to its semantics, may or may not require a data access.

Cache performance - 3

Miss rate can be measured with cache simulators that take an address trace of the instruction and data references, simulate the cache behavior to assert which references hit or miss and report in the end their totals. Many microprocessors today provide hardware to count both the number of misses and of memory references in real time, which turn the task easier.

The variable *miss penalty* is more difficult to estimate. The memory behind the cache may be busy at the time of the miss because of prior memory requests, of a memory refresh, or even of I/O transfers that are taking place. The number of clock cycles also varies at interfaces between different clocks of the processor, the bus or the memory. So, using a single value for *miss penalty* is a simplification.

Finally, since in many cases *miss rate* and *miss penalty* have different values for read and write operations, memory stall cycles could be defined taking this fact into consideration

$$\text{memory stall clock cycles} = \text{instruction count} \cdot \sum_{i=r,w} \frac{\text{operations}(i)}{\text{instruction}} \cdot \text{miss rate}(i) \cdot \text{miss penalty clock cycles}(i) \ .$$

Cache performance - 4

Assume a computer system where the value of *clock cycles per instruction* (CPI) is 1.0 when all memory accesses hit the cache. The only data accesses are load and store instructions which account for 50% of all executed instructions on the benchmark being run. The *miss penalty clock cycles* is 25 and the *miss rate* is 2%.

How much slower is it running compared to a computer system having the same processor and executing the same program with no cache misses, or alternatively, how much faster is it running compared to a computer system having the same processor and executing the same program with 100% cache misses?

0% miss rate

$$\begin{aligned} \text{CPU}_{0\,\text{mr}}\ \text{exec time} &= (\text{CPU clock cycles} + \text{mem stall clock cycles}) \cdot \\ &\quad \cdot \text{clock cycle time} = \\ &= (\text{instruction count} \cdot \text{CPI} + 0) \cdot \text{clock cycle time} = \\ &= 1.00 \cdot \text{instruction count} \cdot \text{clock cycle time} \end{aligned}$$

Cache performance - 5

2% miss rate

$$\text{mem stall clock cycles}_{2\,\text{mr}} = \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot$$

$$\cdot \text{miss rate} \cdot \text{miss penalty clock cycles} =$$

$$= \text{instruction count} \cdot (1.0 + 0.5) \cdot 0.02 \cdot 25 =$$

$$= 0.75 \cdot \text{instruction count}$$

$$\text{CPU}_{2\,\text{mr}}\ \text{exec time} = (\text{CPU clock cycles} + \text{mem stall clock cycles}) \cdot$$

$$\cdot \text{clock cycle time} =$$

$$= \text{instruction count} \cdot (1.00 + 0.75) \cdot \text{clock cycle time} =$$

$$= 1.75 \cdot \text{instruction count} \cdot \text{clock cycle time}$$

Performance ratio

$$\frac{\text{CPU}_{0\,\text{mr}}\ \text{exec time}}{\text{CPU}_{2\,\text{mr}}\ \text{exec time}} = \frac{1.00 \cdot \text{instruction count} \cdot \text{clock cycle time}}{1.75 \cdot \text{instruction count} \cdot \text{clock cycle time}} = 0.571 \ .$$

Cache performance - 6

100% miss rate

$$\text{mem stall clock cycles}_{100\,\text{mr}} = \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot$$

$$\cdot \text{miss rate} \cdot \text{miss penalty clock cycles} =$$

$$= \text{instruction count} \cdot (1.0 + 0.5) \cdot 1.0 \cdot 25 =$$

$$= 37.5 \cdot \text{instruction count}$$

$$\text{CPU}_{100\,\text{mr}}\ \text{exec time} = (\text{CPU clock cycles} + \text{mem stall clock cycles}) \cdot$$

$$\cdot \text{clock cycle time} =$$

$$= \text{instruction count} \cdot (1.00 + 37.5) \cdot \text{clock cycle time} =$$

$$= 38.5 \cdot \text{instruction count} \cdot \text{clock cycle time}$$

Performance ratio

$$\frac{\text{CPU}_{100\,\text{mr}}\ \text{exec time}}{\text{CPU}_{2\,\text{mr}}\ \text{exec time}} = \frac{38.5 \cdot \text{instruction count} \cdot \text{clock cycle time}}{1.75 \cdot \text{instruction count} \cdot \text{clock cycle time}} = 22.0 \ .$$

Cache performance - 7

As this example illustrates, cache behavior can have an enormous impact on performance. Cache misses have a double-barrelled impact on a processor with low CPI and a fast clock

- the lower the CPI is, the higher is the relative impact of the processor independent miss penalty measured as a fixed number of clock cycles
- even if the memory hierarchies for two computer systems are identical, the processor with the higher clock rate has a larger number of memory stall cycles.

The importance of the cache for processors with low CPI and high clock rates is thus greater and, therefore, greater is also the danger of neglecting cache behavior in assessing the performance of these computer systems.

Cache performance - 8

Some designers prefer defining *miss rate* as the number of misses per instruction, rather than the number of misses per memory access. These two parameters are related by

$$\frac{\text{total number of misses}}{\text{instruction count}} = \text{miss rate} \cdot \frac{\text{memory accesses}}{\text{instruction count}} \ .$$

The advantage of using *misses per instruction* to specify the *miss rate* is that it converts this figure of merit into a value which is implementation independent. Speculative processors, for instance, fetch about twice as many instructions as they are actually committed. Thus, reducing artificially the miss rate if measured as misses per memory reference.

The drawback is that *misses per instruction* is architecture dependent, which means it does not make any sense using it to compare two quite different architectures such as MIPS and Intel 80x86.

Cache performance - 9

Instead of concentrating on the *miss rate* to evaluate the performance of the memory hierarchy, one could use as a better figure of merit the *average memory access time*

$$\text{average memory access time} = \text{hit time} + \text{miss rate} \cdot \text{miss penalty clock cycles} \cdot \text{clock cycle time} .$$

The variable *hit time* is the time to access the cache on a cache hit and the other variables, *miss rate* and *miss penalty*, have the same meaning as before.

Cache performance - 10

A decision must be taken on whether implementing a 16 KB instruction cache and a 16 KB data cache versus a 32 KB unified cache for a particular computer system.

Simulation results on properly chosen benchmarks have indicated the following values for the number misses per 1000 instructions: 3.82 (16 KB instruction cache), 40.9 (16 KB data cache) and 43.3 (32 KB unified cache), where 36% of the totality of the instructions that were executed are data transfer. Assume that a hit takes 1 clock cycle and the miss penalty clock cycles is 200. Assume also that a load or store hit takes 1 extra clock cycle on the unified cache since there is a single port to service two simultaneous requests and that the data cache and the unified cache implement a write-through policy with a write buffer (stalls on writing can be ignored).

$$\text{miss rate} = \frac{\dfrac{\text{misses per 1000 instructions}}{1000}}{\dfrac{\text{memory accesses}}{\text{instruction count}}}$$

Cache performance - 11

The *instruction cache* has exactly one memory access per instruction

$$\text{miss rate}_{16\,\text{KB instruction}} = \frac{3.82 \cdot 10^{-3}}{1.00} = 0.004 \quad .$$

The *data cache* has in average 0.36 memory accesses per instruction

$$\text{miss rate}_{16\,\text{KB data}} = \frac{40.90 \cdot 10^{-3}}{0.36} = 0.114 \quad .$$

The *unified cache* has in average 1.36 memory accesses per instruction

$$\text{miss rate}_{32\,\text{KB unified}} = \frac{43.30 \cdot 10^{-3}}{1.36} = 0.032 \quad .$$

Cache performance - 12

The effective *miss rate* of the combined *instruction + data caches* is given by

$$\text{miss rate}_{16\,\text{KB instruction + data}} = \frac{\text{mem accesses}_{\text{intruction}}}{\text{mem accesses}_{\text{total}}} \cdot \text{miss rate}_{16\,\text{KB intruction}} +$$

$$+ \frac{\text{mem accesses}_{\text{data}}}{\text{mem accesses}_{\text{total}}} \cdot \text{miss rate}_{16\,\text{KB data}} =$$

$$= \frac{1.00}{1.36} \cdot 0.004 + \frac{0.36}{1.36} \cdot 0.114 = 0.033 \quad .$$

Notice that the 32 KB unified cache has a slightly lower miss rate than the combination of two separate 16 KB instruction and data caches!

Cache performance - 13

The *average memory access clock cycles* is given by

$$\text{average memory access clock cycles} =$$

$$= \frac{\text{mem accesses}_{\text{intruction}}}{\text{mem accesses}_{\text{total}}} \cdot \left(\text{hit time} + \text{miss rate}_{\text{instruction}} \cdot \text{miss penalty clock cycles}\right) +$$

$$+ \frac{\text{mem accesses}_{\text{data}}}{\text{mem accesses}_{\text{total}}} \cdot \left(\text{hit time} + \text{miss rate}_{\text{data}} \cdot \text{miss penalty clock cycles}\right)$$

yielding in each case

$$\text{average memory access clock cycles}_{\text{32 KB unified}} =$$

$$= \frac{1.00}{1.36} \cdot \left(1.0 + 0.032 \cdot 200\right) + \frac{0.36}{1.36} \cdot \left(2 + 0.032 \cdot 200\right) = 7.66$$

Cache performance - 14

$$\text{average memory access clock cycles}_{\text{16 KB instruction + data}} =$$

$$= \frac{1.00}{1.36} \cdot (1.0 + 0.004 \cdot 200) + \frac{0.36}{1.36} \cdot (1 + 0.114 \cdot 200) = 7.62 \ .$$

Although the two separate 16 KB instruction and data caches have a slightly higher effective miss rate than the 32 KB unified cache, the fact is that when one considers the average memory access time, the result is reversed due to the existence in this case of two memory ports per clock cycle, thus avoiding the structural hazard present at the single-port unified cache.

Cache performance - 15

What is the impact of two different cache organizations on the performance of a specific processor?

Assume that the CPI with a perfect cache (no cache misses) is 1.6, that the clock cycle is 0.35 ns and that there are 1.4 memory references per instruction for the benchmark being run. The size of both caches is 128 KB and both have a block size of 64 bytes. The first is organized as direct mapped and the second as 2-way set associative. As the Opteron data cache illustrates, a multiplexor must be added to select between the data in the set depending on the tag match. Since the speed of the processor can be tied directly to the speed of a cache hit, suppose the processor clock cycle time is stretched 1.35 times to accomodate the selection multiplexor of the set associative cache. In both cases, the hit time is 1 clock cycle, the miss penalty is 65 ns and the miss rate is 2.1% and 1.9%, respectively, for the direct map and the 2-way set associative caches.

Cache performance - 16

$$\text{average memory access time}_{\text{128 KB direct mapped}} =$$

$$= \text{hit clock cycles} \cdot \text{clock cycle time} + \text{miss rate} \cdot \text{miss penalty} =$$

$$= 1.0 \cdot 0.35 + 0.021 \cdot 65 = 1.72\,\text{ns}$$

$$\text{average memory access time}_{\text{128 KB 2-way set associative}} =$$

$$= \text{hit clock cycles} \cdot \text{effective clock cycle time} + \text{miss rate} \cdot \text{miss penalty} =$$

$$= 1.0 \cdot 1.35 \cdot 0.35 + 0.019 \cdot 65 = 1.71\,\text{ns}$$

Notice that the 128 KB 2-way set associative cache has a slightly lower average memory access time than the 128 KB direct mapped cache!

Cache performance - 17

$$\text{CPU execution time}_{\text{128 KB direct mapped}} =$$

$$= \text{instruction count} \cdot \text{CPI} \cdot \text{clock cycle time} +$$

$$+ \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot \text{miss penalty} =$$

$$= (1.6 \cdot 0.35 + 1.4 \cdot 0.021 \cdot 65) \cdot \text{instruction count} = 2.47 \cdot \text{instruction count}$$

$$\text{CPU execution time}_{\text{128 KB 2-way set associative}} =$$

$$= \text{instruction count} \cdot \text{CPI} \cdot \text{effective clock cycle time} +$$

$$+ \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot \text{miss penalty} =$$

$$= (1.6 \cdot 1.35 \cdot 0.35 + 1.4 \cdot 0.019 \cdot 65) \cdot \text{instruction count} = 2.49 \cdot \text{instruction count}$$

Although the 128 KB 2-way set associative cache has a lower average memory access time than the 128 KB direct mapped cache, the fact is that when one considers the CPU execution time, the result is reversed due to the stretching of the clock cycle in the case of the associative cache.

Cache performance - 18

For an *out-of-order execution* processor, *miss penalty* can no longer be defined as the full latency of the miss to memory. This question is relevant since *out-of-order* processors are known to tolerate some latency due to cache misses without affecting its performance.

The number of *memory stall cycles* can be redefined to lead to a new definition of *miss penalty* as a non overlapped latency

$$\text{memory stall cycles} = \text{instruction count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot (\text{total miss penalty} - \text{overlapped miss latency}) .$$

It is said that a *processor is stalled in a clock cycle* if it does not retire the maximum possible number of instructions in that cycle. The *stall* is attributed to the first instruction that could not be retired. *Latency*, on the other hand, can be measured from the time the memory instruction is queued in the instruction window, or from the time the address is generated, up to the time the data transfer takes place. Any alternative is valid as long as it is used consistently.

Cache performance - 19

Consider that the processor of the last example has a 1.35 times longer clock cycle time to support out-of-order execution and has a direct mapped cache. Assume also that 30% of the 65 ns miss penalty can be overlapped, thus reducing the average memory stall cycle to 45.5 ns.

$$\text{average memory access time}_{\text{direct mapped, OOO}} =$$

$$= \text{hit clock cycles} \cdot \text{clock cycle time} + \text{miss rate} \cdot \text{effective miss penalty} =$$

$$= 1.0 \cdot 1.35 \cdot 0.35 + 0.021 \cdot 45.5 = 1.43 \text{ ns}$$

$$\text{CPU execution time}_{\text{direct mapped, OOO}} =$$

$$= \text{instruction count} \cdot \text{CPI} \cdot \text{clock cycle time} =$$

$$+ \text{instruc count} \cdot \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{miss rate} \cdot \text{effective miss penalty} =$$

$$= \left(1.6 \cdot 1.35 \cdot 0.35 + 1.4 \cdot 0.021 \cdot 45.5\right) \cdot \text{instruc count} = 2.09 \cdot \text{instruc count}$$

However, due to its complexity, designers tend to use simulators of the out-of-order processor and of the memory when evaluating the trade-offs in the memory hierarchy to assess that an improvement that helps the average memory latency, also helps the program performance.

Cache optimization - 1

The traditional approach to improve the behavior of a cache is to minimize the miss rate. Misses can be modelled into three basic categories

- *compulsory misses* – misses that occur even if the cache had an infinite size; the very first access to any byte or word will always translate into a miss because the memory block where the byte or word resides must be brought first into the cache; they are also known as *cold-start misses* or *first-reference misses*
- *capacity misses* – misses that occur in a fully associative cache; they are directly related to the cache size, if the cache is not large enough, memory blocks will be discarded during program execution and later will be retrieved because they are needed again
- *conflict misses* – misses that occur specifically due to the internal organization of the cache; putting aside the case of a fully associative cache and considering the direct mapped cache as an instance of an 1-way set associative cache, memory blocks may be discarded and later retrieved simply because too many blocks are mapped in some of the sets; they are also known as *collision misses*.

Cache optimization - 2

Miss rate for the Alpha architecture (DEC) using 10 SPEC2000 benchmarks (5 SPECint2000 and 5 SPECfp2000) – LRU replacement – block size = 64 bytes

Source: adapted from Computer Architecture: A Quantitative Approach

			Miss rate components (*value / relative % of total*)					
Cache size (KB)	***Degree of Associativity***	***Total miss Rate***	***Compulsory***		***Capacity***		***Conflict***	
4	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
16	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
64	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
256	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%

Cache optimization - 3

As it should be expected *compulsory misses* are independent of the cache size, *capacity misses* decrease as the cache size increases and *conflict misses* decrease as the degree of associativity increases.

Enlarging the cache size is important. If the upper level memory is too small to contain the fraction of the program's code and data needed by the locality principle, then a significant portion of time is spent transferring memory blocks between two adjacent levels of the hierarchy and *thrashing* will occur. Hence, the computer system will run closer to the speed of the lower-level memory, or even slower due to miss overhead. On the other hand, implementing full associativity to get rid of the conflict misses is expansive in terms of hardware and may lead to a slow clock rate, thus lowering the overall performance.

The *miss model* just presented has its own limits: it gives an insight into average behavior, but does not explain individual misses, or incorporates the replacement policy. Many techniques that reduce miss rates, also increase the *hit time* and/or the *miss penalty*. Therefore, one needs a balanced approach to make the whole computing system faster.

Cache optimization - 4

A straight way to reduce the miss rate is ***to increase the block size***. Larger block sizes will diminish *compulsory misses* due to spatial locality. However, if the block size is too large relative to the cache size, the reduction on the number of cache lines tends to increase *capacity* and *conflict misses* and to make the total miss rate worse.

Miss rate vs. block size for DECStation 5000 using SPEC92 benchmarks

Source: Gee, Hill, Pnevimatikatos, Smith, "Cache performance of the SPEC92 benchmark suite", IEEE Micro 13:4, August 1993

	Cache size			
Block size (bytes)	**4 *KB***	**16 *KB***	**64 *KB***	**256 *KB***
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	**7.00%**	**2.64%**	1.06%	0.51%
128	7.78%	2.77%	**1.02%**	**0.49%**
256	9.51%	3.39%	1.15%	**0.49%**

Cache optimization - 5

At the same time, larger block size tend to increase the *miss penalty*, because of the number of bytes to be transferred also increases

$$\text{miss penalty} = \text{cache latency clock cycles} \cdot \text{clock cycle time} + \\ + \text{block size} / \text{bandwidth (clock cycles)} \cdot \text{clock cycle time} \ .$$

Average memory access clock cycles vs. block size for DECStation 5000 using SPEC92 benchmarks

Source:Computer Architecture: A Quantitative Approach

hit clock cycles = 1

cache latency clock cycles = 80

bandwidth = 8 bytes / clock cycle

		Cache size			
***Block size* (*bytes*)**	***Miss penalty* (*clock cycles*)**	**4 *KB***	**16 *KB***	**64 *KB***	**256 *KB***
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

Cache optimization - 6

Since the the choice of the block size will be affected by the latency and the bandwidth of the lower level memory, the cache designer has to consider both the miss rate and the miss penalty when deciding on the cache block size.

High latency and high bandwidth encourage a large block size because the cache receives many more bytes per miss for a small increase of the miss penalty. On the other hand, low latency and low bandwidth encourage smaller block sizes because the time saved from a larger block is not significative. One has to remember that having a large number of small blocks may reduce conflict misses.

Capacity misses are obviously reduced by increasing the cache size. However, there is a limit to it because the hit time is potentially longer due to the increase of the logic complexity, making it also both a higher cost and a higher power solution. Notwithstanding, his technique has been specially popular in caches located off the processor chip.

Cache optimization - 7

In the same way, although higher associativity reduces the miss rate, the average memory access time is not always also lessened.

Average memory access clock cycles vs. associativity for DECStation 5000 using SPEC92 benchmarks – LRU replacement – block size = 64 bytes

Source:Computer Architecture: A Quantitative Approach

$$\text{hit clock cycles} = 1$$

$$\text{clock cycle time}_{2-way} = 1.36 \cdot \text{clock cycle time}_{1-way}$$

$$\text{clock cycle time}_{4-way} = 1.44 \cdot \text{clock cycle time}_{1-way}$$

$$\text{clock cycle time}_{8-way} = 1.52 \cdot \text{clock cycle time}_{1-way}$$

	Associativity			
***Cache size* (KB)**	**1-*way***	**2-*way***	**4-*way***	**8-*way***
16	2.23	2.40	2.46	2.53
32	2.06	2.30	2.37	2.45
64	1.92	2.14	2.18	2.25
128	1.52	1.84	1.92	2.00
256	1.32	1.66	1.74	1.82

Main memory

Main memory satisfies primarily the demands of caches and serves as the I/O interface both for the swapping area in a virtual memory organization and for the different device controllers, acting as a destination for input data and a source for output data.

Traditionally, *memory latency*, which affects the cache miss penalty, is the primary concern of the cache, while *memory bandwidth* is the primary concern of the I/O controllers, but it is also important for multilevel caches with their larger block sizes. Higher bandwidth can be achieved by using multiple memory banks, by widening the data bus and by introducing *burst transfer mode*, where multiple words are transferred in the same access.

Main memory is built around DRAM chips. Presently, *memory latency* is expressed through two figures of merit

- *access time* – time interval between the issue of a read / write request and the instant the associated data becomes available / is stored
- *cycle time* – minimum time interval between unrelated memory requests.

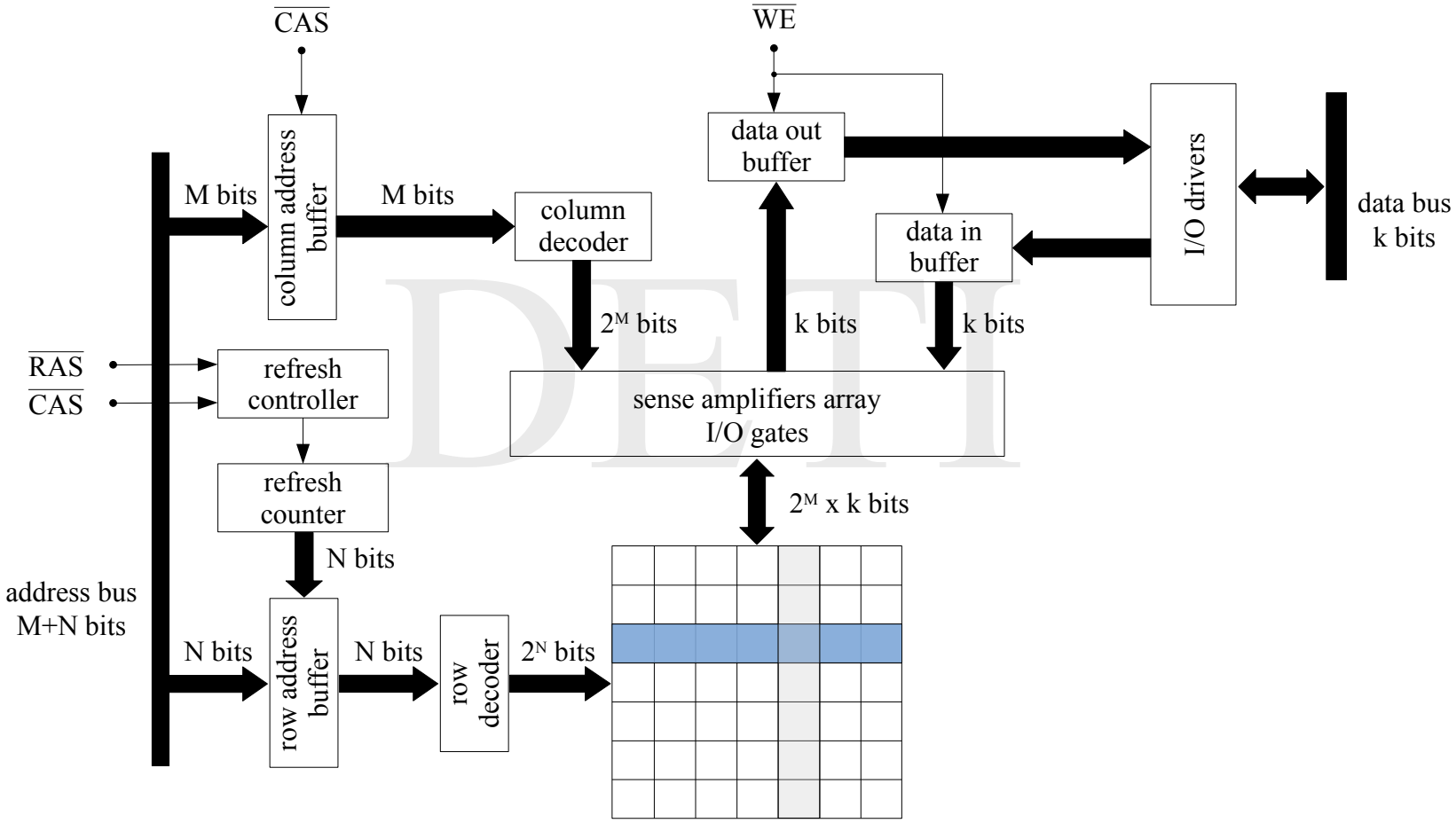
DRAM

As dynamic RAMs grew in capacity, the package cost with all the required address pins became an issue. In order to solve it, the address lines were multiplexed: part of the address is latched internally during the first phase of memory access, with the *row access strobe* control signal, and the remaining address is latched later, during the second phase, with the *column access strobe* control signal.

In order to pack more bits per chip, a single transistor is used to store one bit. Reading this bit destroys the stored information, which means it has to be restored (written back) after each reading. This is one of the reasons why the memory cycle time is longer than the memory access time. With the introduction of multiple bank DRAMs, which enable the rewrite portion of the cycle to be hidden, this problem has been attenuated.

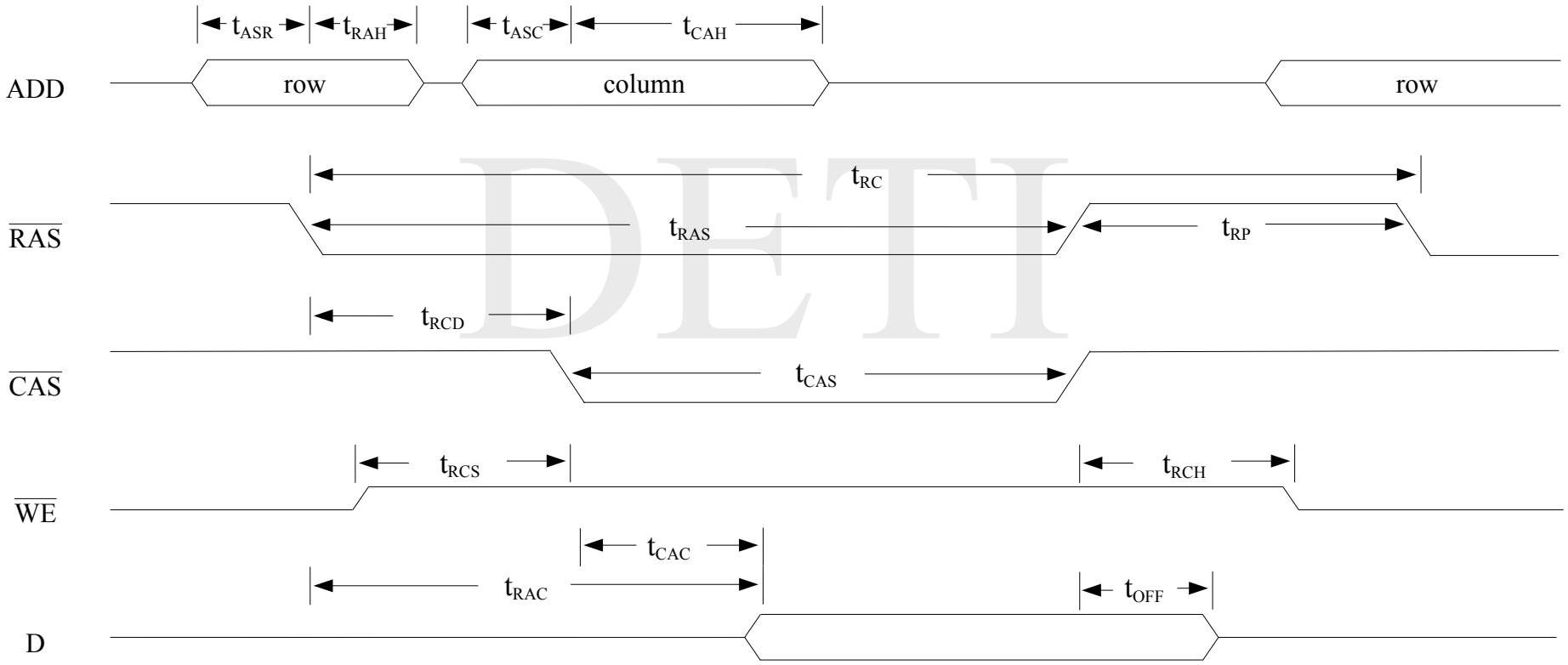
Also, to prevent loss of information when a stored bit is not read for a long period of time, a charge refreshment has to take place. All the bits belonging to the same row can be refreshed simultaneously just by acessing a column of that row. Hence, means have to be provided for every DRAM to access each of its rows within some time frame, typically a few tens of milliseconds. Memory controllers have hardware to carry out this task.

Asynchronous DRAM - 1



Asynchronous DRAM - 2

Read operation



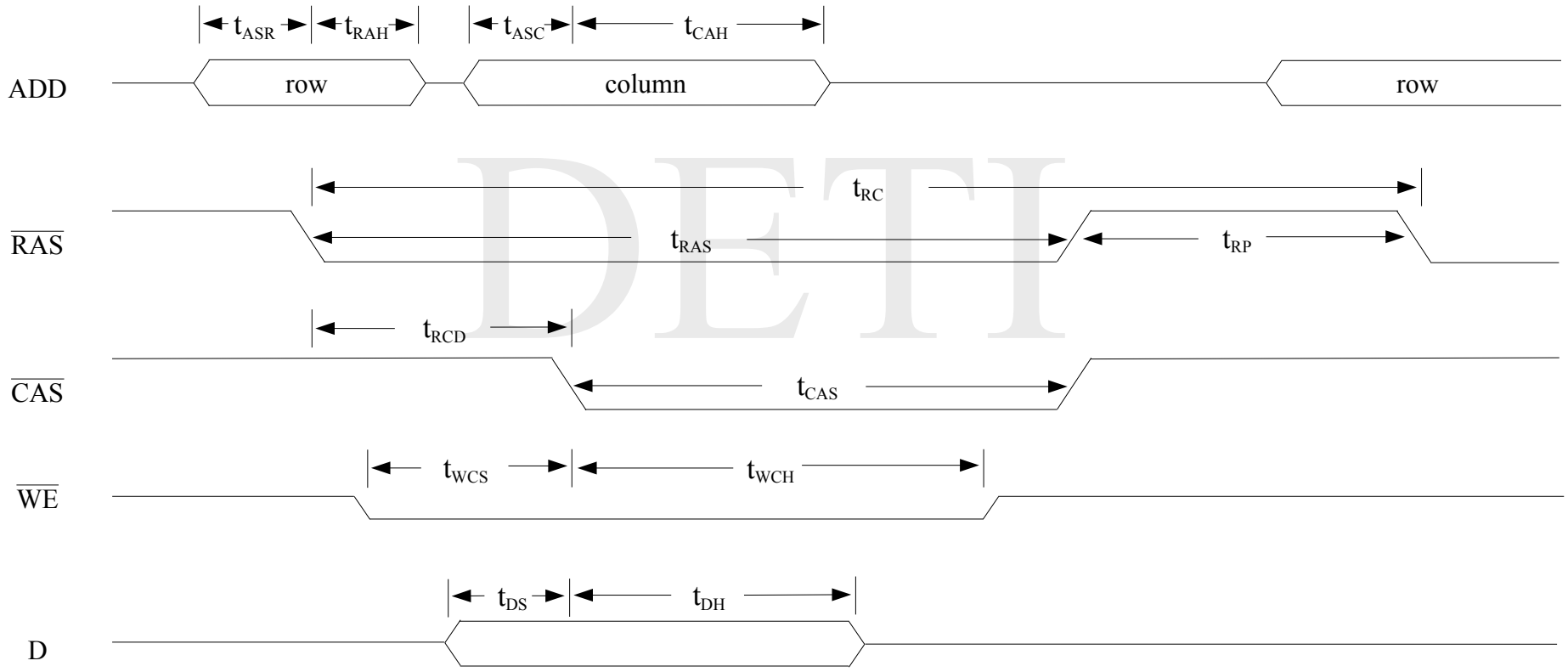
Asynchronous DRAM - 3

Read operation

- the *write enable* signal is asserted high by the *memory controller* to mean a read operation is taking place
- a N-bit address is placed in the address bus and is latched into the *row address buffer* upon the falling edge of the *row access strobe* asserted by the *memory controller*
- the data values stored in the selected row of memory cells are then sensed and maintained in the *sense amplifiers array* to be later on written back to the memory cells
- a M-bit address is placed next in the address bus and is latched into the *column address buffer* upon the falling edge of the *column access strobe* asserted by the *memory controller*
- the data values kept in the selected column of the *sense amplifiers array* is latched into the *data out buffer* to drive the data bus

Asynchronous DRAM - 4

Write operation



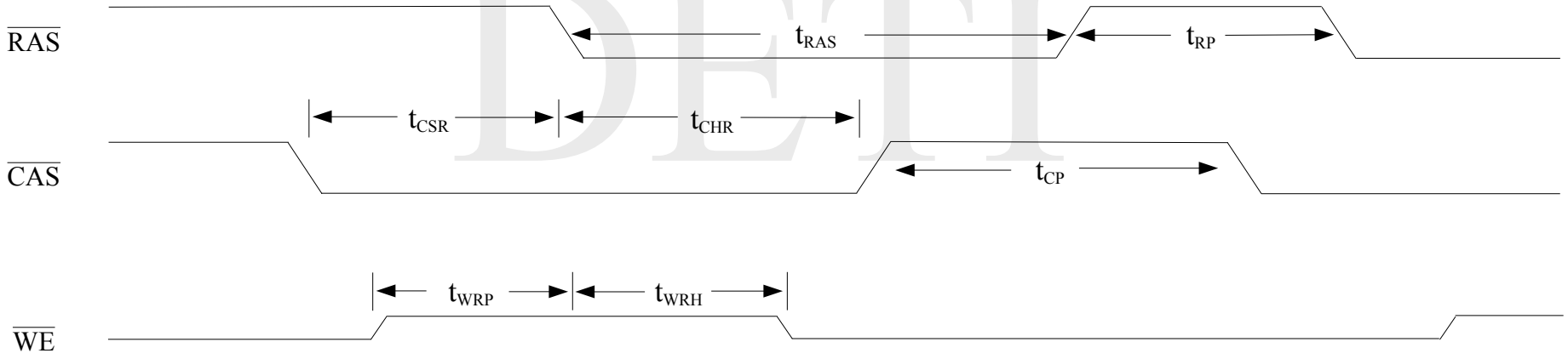
Asynchronous DRAM - 5

Write operation

- the *write enable* signal is asserted low by the *memory controller* to mean a write operation is taking place
- a N-bit address is placed in the address bus and is latched into the *row address buffer* upon the falling edge of the *row access strobe* asserted by the *memory controller*
- the data values stored in the selected row of memory cells are then sensed and maintained in the *sense amplifiers array* to be later on written back to the memory cells
- a M-bit address is placed next in the address bus and is latched into the *column address buffer* upon the falling edge of the *column access strobe* asserted by the *memory controller*
- the data values present in the data bus are latched into the *data in buffer* upon the falling edge of the *column access strobe* and replace the values stored in the selected column of the *sense amplifiers array* before the writing back to the memory cells

Asynchronous DRAM - 6

Refresh operation (CAS before RAS)



Asynchronous DRAM - 7

Refresh operation (CAS before RAS)

- the *column access strobe* signal is asserted low by the *memory controller* before the *row access strobe* and the *write enable* signal is asserted high, the control logic inside the chip interprets this arrangement as a refresh operation
- address lines inputs are ignored and the contents of the *refresh counter* is latched into the *row address buffer* upon the falling edge of the *row access strobe*
- the data values stored in the selected row of memory cells are then sensed and maintained in the *sense amplifiers array* to be later on written back to the memory cells

Asynchronous DRAM - 8

Asynchronous DRAM interface was modified to improve the performance of read and write operations to the same memory row by avoiding the requirement of pre-charging and opening the row repeatedly for access to a different column.

In *page mode* DRAM, after a row is opened by holding *row access strobe* low, the row is kept open and multiple read or write operations are performed to any of the columns of the row. Each column access is started by asserting *column access strobe* low and presenting a column address. For read operations, the *write enable* signal should also be set high. For write operations, the *write enable* signal should be set low and data values to be written should be presented along with the column address.

Page mode DRAM was later improved with a small modification which further reduced latency, giving rise to the so-called *fast page mode* DRAM, or FPM DRAM. In a *page mode* DRAM, *column access strobe* is asserted after the column address is supplied. In a FPM DRAM, the column address can be supplied while *column access strobe* is still deasserted. The column address propagates through the column address data path, but does not output data on the data pins until *column access strobe* is asserted.

Synchronous DRAM - 1

Synchronous DRAM, or SDRAM, changes in a radical way how the external memory interface interacts with the device. Control signals no longer have a direct effect on the internal functions, but an externally controlled clock signal is used to manage a built-in finite state machine which acts upon incoming commands. Thus, $\overline{\texttt{RAS}}$ and $\overline{\texttt{CAS}}$ signals no longer act as strobes, but become instead part of the input command.

The internal operations are pipelined to improve performance. Previously initiated operations being completed, while new commands are received and start processing.

The two-dimensional memory cell array is divided into several equally sized, but independent sections, called *banks*, allowing the device to operate on a memory access command, read or write, in each bank concurrently and speed up access in an interleaved manner.

Burst transfer mode is also implemented for accessing multiple data words with the same command.

Synchronous DRAM - 2

Later generations of SDRAMs improved transfer bandwidth by allowing data transfer to take place both in the rising edge and the falling edge of the externally controlled clock signal. This optimization is known as *double data rate*, or DDR.

DDR has given rise to a sequence of standards

- DDR, or DDR1 – has a voltage supply of 2.5 V and operates at clock frequencies of 133, 150 and 200 MHz
- DDR2 – has a voltage supply of 1.8 V and operates at clock frequencies of 266, 333 and 400 MHz
- DDR3 – has a voltage supply of 1.5 V and operate at clock frequencies of 533, 666 and 800 MHz
- DDR4 – has a voltage supply of 1.2 V and operate at clock frequencies of 1066, 1333 and 1600 MHz.

DRAMs are usually sold in small boards, called *dual inline memory modules*, or DIMMs, which contain 4 to 16 SDRAM chips and are normally organized to provide a word length of 64 data bits + *error correcting code* bits.

Synchronous DRAM - 3

64 Mbit SDRAM

(organized in 4 banks of 2048 rows x 1024 columns memory cells of 8 bits each)

$\overline{CS}$	$\overline{RAS}$	$\overline{CAS}$	$\overline{WE}$	BAn	A10	An	Command
H	X	X	X	X	X	X	command inhibit (the device is not selected)
L	H	H	H	X	X	X	no operation
L	H	H	L	X	X	X	burst terminate: stop a burst read, or a burst write, in progress
L	H	L	H	bank	L	column	read a burst of data from the current active row
L	H	L	H	bank	H	column	read a burst of data from the current active row with precharge (close the row) when done
L	H	L	L	bank	L	column	write a burst of data from the current active row
L	H	L	L	bank	H	column	write a burst of data from the current active row with precharge (close the row) when done
L	L	H	H	bank	row		activate (open) the row for a read or write command
L	L	H	L	bank	L	X	precharge, or deactivate (close), the current row of selected bank
L	L	H	L	X	H	X	precharge, or deactivate (close), the current row of all banks
L	L	L	H	X	X	X	refresh one row of each bank (auto refresh) using an internal counter (all rows must be precharged)
L	L	L	L	00	mode		load mode register for chip configuration: the control word is in A10-A0 the most significant settings are to program CAS latency (2 or 3 clock cycles) and burst transfer length (1, 2, 4 or 8 words)

DRAM generations

Time specifications

Source: Computer Architecture: A Quantitative Approach

Production year	Chip size (bit)	DRAM type	Slowest DRAMs (ns)	Fastest DRAMs (ns)	CAS / data transfer time (ns)	Cycle time (ns)
1980	64K	DRAM	180	150	75	250
1983	256K	DRAM	150	120	50	220
1986	1M	DRAM	120	100	25	190
1989	4M	DRAM	100	80	20	165
1992	16M	DRAM	80	60	15	120
1996	64M	SDRAM	70	50	12	110
1998	128M	SDRAM	70	50	10	100
2000	256M	DDR1	65	45	7	90
2002	512M	DDR1	60	40	5	80
2004	1G	DDR2	55	35	5	70
2006	2G	DDR2	50	30	2.5	60
2010	4G	DDR3	36	28	10	37
2012	8G	DDR3	30	24	0.5	31

DDR-SDRAM classification

DIMM characterization

Source: Computer Architecture: A Quantitative Approach

Standard	Clock rate (MHz)	M (transf/s)	DRAM name	MB/s/DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10664	PC10700
DDR3	800	1600	DDR3-1600	12800	PC12800
DDR4	1066-1600	2133-3200	DDR4-3200	17056-25600	PC25600

Suggested reading

- *Computer Architecture: A Quantitative Approach*, Hennessy J.L., Patterson D.A., 6th Edition, Morgan Kaufmann, 2017
  - Appendix B: *Review of Memory Hierarchy*
  - Chapter 2: *Memory Hierarchy Design*
- *Computer Organization and Architecture: Designing for Performance*, Stallings W., 10th Edition, Pearson Education, 2016
  - Chapter 4: *Cache Memory*
  - Chapter 5: *Internal Memory*
  - Chapter 6: *External Memory*
  - *Chapter* 8: *Operating System Support*