

Basic concepts in CACHING

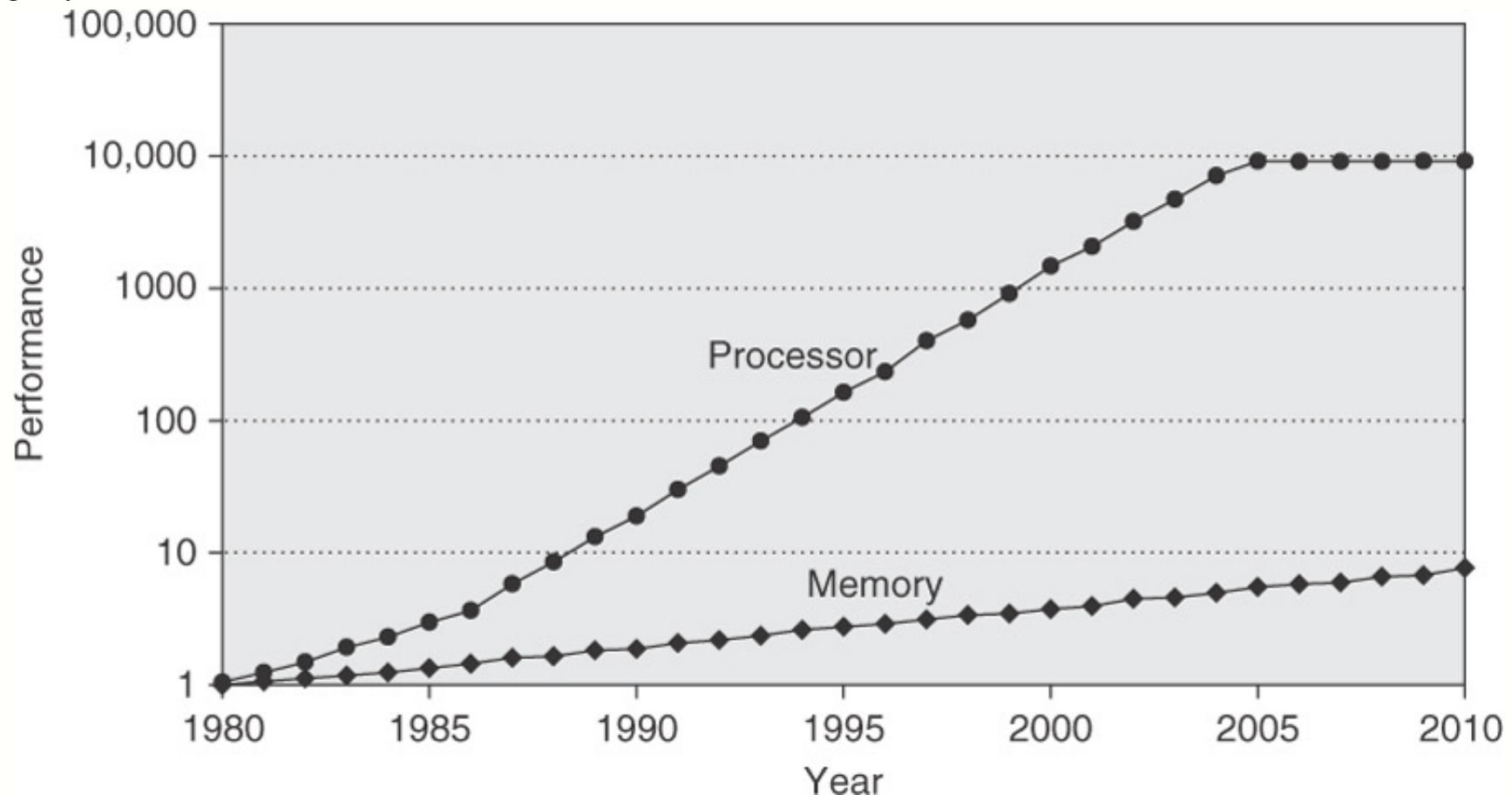
- Cache basic operation
- Direct-Mapped and Set-Associative cache
- Multiple-level caching

CPU and main memory

- Data exchange between CPU and RAM is a critical issue for computer performance.
- On a historical basis, since the very first computer, main memory has always been slower than the CPU, and the gap has widened in time.
- Moreover, data and instructions between CPU and RAM have to transit through the bus, which introduces a further delay.

CPU and main memory

- Performance ratio of CPU vs RAM in time, with ratio 1 in 1980 (Hennessy-Patterson, Fig. 5.2). RAM increase: $\sim 7\%$ per year. CPU: $\sim 25\%$ per year until 1986, $\sim 52\%$ until 2000, $\sim 52\%$ up to 2005, 0% later.



CPU and main memory

- Clearly, it is useless building ever speedier and sophisticated processors, if instructions to be executed and data to be worked on cannot be fetched as quickly.
- In datapaths considered so far, the assumption was that the Instruction Memory and the Data Memory would run at the same speed as the other pipeline components.
- This assumption actually requires a very complex system to manage information flow between CPU and RAM.

CPU and main memory

- The basic idea to overcome the problem of a slow RAM is using a *hierarchy of memories*, each level speedier (and more expensive) and smaller, the closer it is to CPU, to feed the CPU with the required data.
- This is the basic technique in caching, applied to all CPUs and to general purpose architectures.

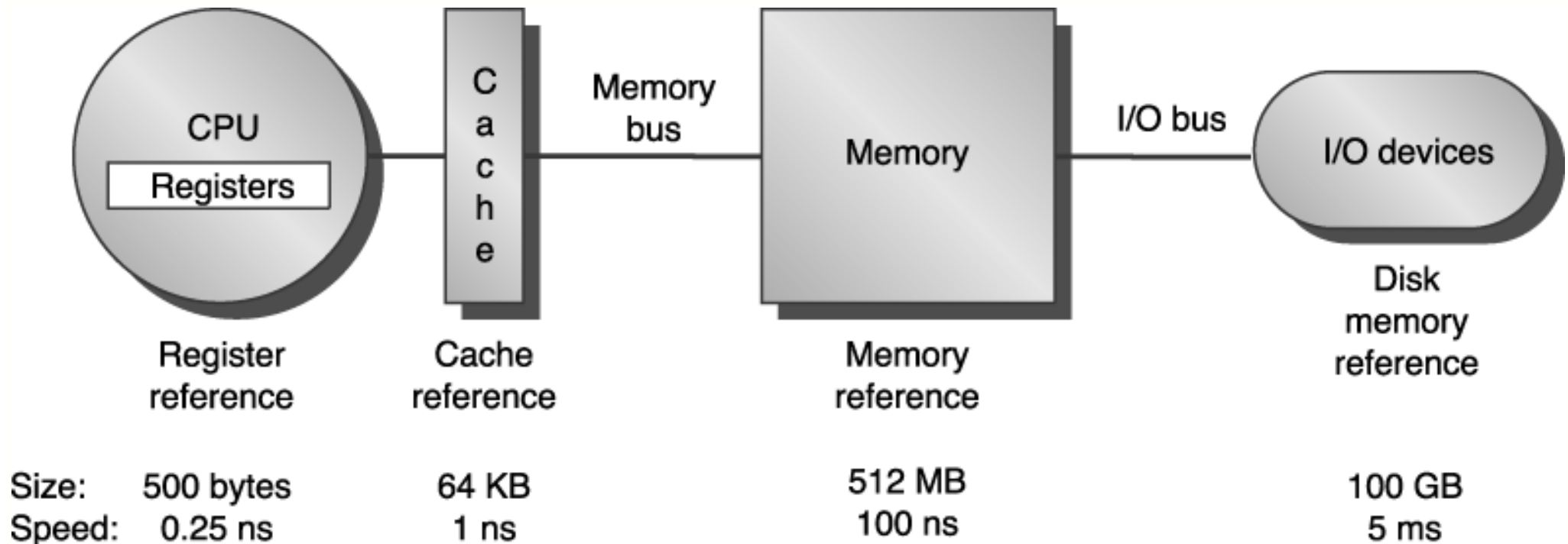
Memory technology	Typical access time	Gbyte cost (in \$) in 2014
SRAM	0.5 – 5 ns	4000 – 10000
DRAM	50 – 70 ns	100 – 200
SSD	0,1 ms	0,45
HDD	2,9 – 12 ms	0,05 – 0,1

Caching

- The caching technique is not typical of the CPU-RAM relationship, it is very basic and applies to the whole memory subsystem in a computer:
 - Registers act as a cache to the actual hw cache
 - The hw cache acts as a cache to RAM
 - RAM acts as a cache to the hard disk (virtual memory)
 - The hard disk acts as a cache to slower magnetic devices

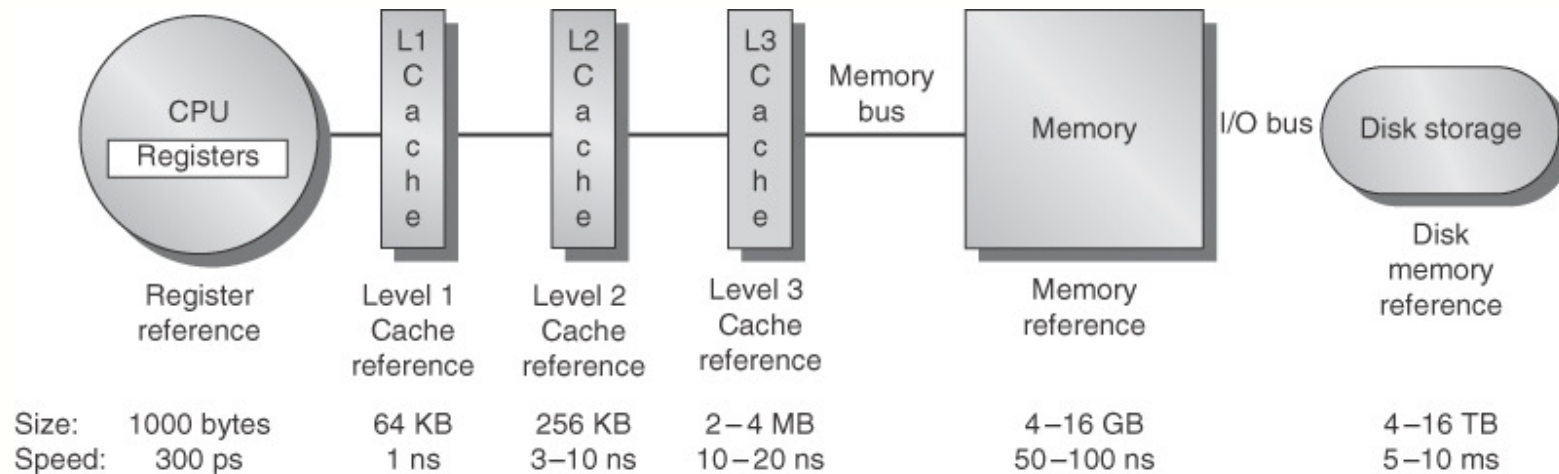
Caching

- Here is a sound relationship among memories, capacities and speed (Hennessy-Patterson, Fig. 5.1):

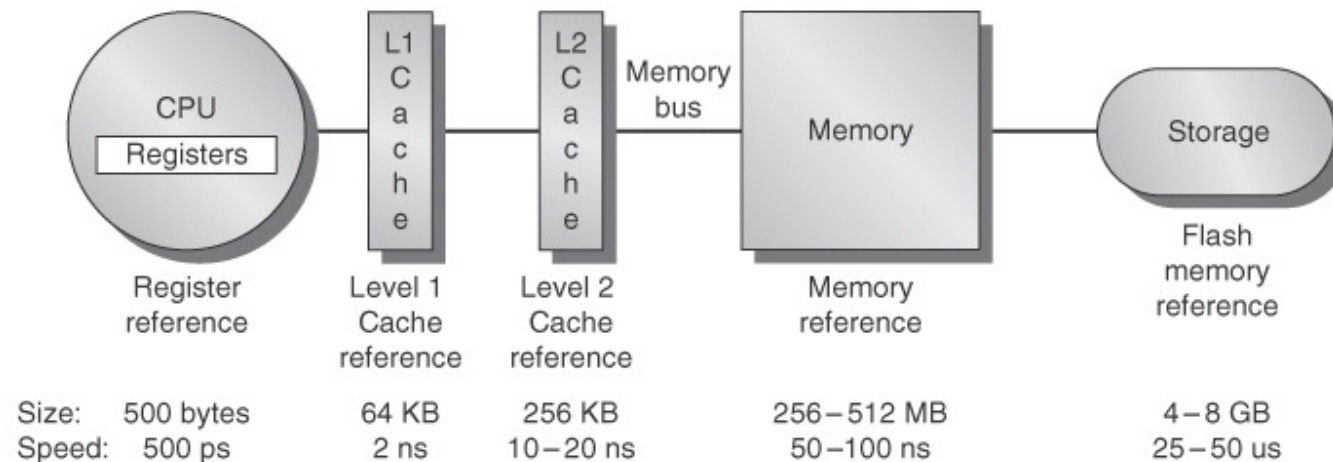


Caching

- A more detailed case: server and PMD (H-P5, Fig. 2.1):



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

Caching

- The concept of the cache was born together with that of computer:
- *Ideally, one would desire an indefinitely large memory capacity such that any particular...word would be immediately available...We are forced to recognize the possibility of constructing a hierarchy of memories, each which has greater capacity than the preceding but which is less quickly accessible*

A. Burks, H. Goldstine, J. von Neumann

Preliminary Discussion of the logical Design of
Electronic Computing Instrument (1946)

Caching

- Even if the cache concept dates as early as the Von Neuman type of computer, only in the late 60s were produced the first CPU equipped with a true cache.
- The basic idea is well understood: a cache is a small, but speedy memory holding part of the data available in RAM.
- If data and instructions required by the CPU are found to a great percentage in the cache, the performance penalty due to slow access to RAM is reduced dramatically.

Caching

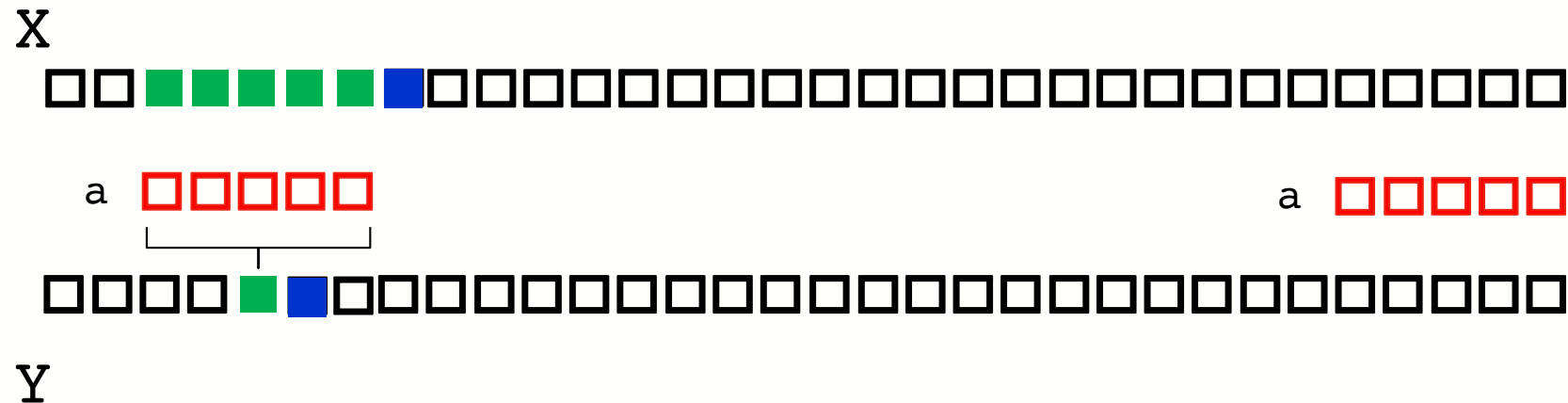
- Current CPUs have separate caches for data and for instructions. With this architecture, in a pipelined datapath, it is possible, within the same clock cycle:
 - to fetch an instruction from the instruction cache (I-cache), that we have so far called the Instruction Memory, and
 - to load/to store data (for a different instruction) from/to the data cache (D-cache), that we have so far called Data Memory.
- I-cache and D-cache together are the **first-level cache** (L1).
- Most processors have a **second-level cache** (L2) as well, and there are even processors with a **third-level cache** (L3).

Cache and locality

- Caches yield great advantages because the *principle of locality* holds in most cases, in two modes:
- **Spatial locality:**
 - Memory items at addresses **near** items just referenced will be referenced in close time (as an instance, instructions are executed in sequence, and array elements are read/written in sequence as well).
- **Temporal Locality:**
 - Memory items **recently** addressed are likely to be referenced again in the near future (e.g. instructions and variables within a loop).

Cache and locality

Example: convolution of signal X with filter **a** into output Y



X & Y **Space** locality: **current** addresses – **next** address

a **Temporal** locality: **same** addresses


Cache basic operation

- *N.B. Here we only consider read operations from RAM. Later we will cover also operations involving a modification of the data.*
- For the purpose of operating with a cache, RAM is split into fixed-size blocks name **cache lines** (or simply **lines**) or **cache blocks** (or simply **blocks**).
 - A block contains a fixed number of consecutive bytes of RAM (4 to 256, according to implementations, always a power of 2)
- Blocks are numbered consecutively starting from 0, so with 32-byte blocks we have:
 - block 0: RAM bytes from 0 to 31
 - block 1: RAM bytes from 32 to 63
 - block 2: RAM bytes from 64 to 95

Cache basic operation

- Each block is identified with its address in RAM, which is the RAM address of the first byte in the block.
- Since blocks contain 2^m bytes, each block address has the following pattern:

x x ... x x 0 0 ... 0 0



- Question: what is the relationship between the number of a block and its address?

Basic cache operation

- At any time, some RAM blocks are in the cache too (we assume first-level only caches, separate for instructions and for data, working at roughly the same speed as the CPU).
- When a word (for an integer, or an instruction in 4 bytes) is addressed, the hardware checks if the word is in the cache too.
- If so, this is a **cache hit** and every thing goes on normally, since the cache can work at the same speed of the other datapath components.
- If not, this is a **cache miss**: (the block containing) the missing data is fetched from RAM. Occasionally, a block must be removed from the cache, to accommodate for the new (missing) one.

Direct-mapped cache

- **Direct-Mapped** caches are the most simple ones. A direct-mapped cache consists of 2^n entries, with consecutive numbers.
- Each entry stores a block: 2^m consecutive bytes from RAM (usually $2^m = 32$ or 64). Each entry has two information with it:
 1. a *validity bit* for the entry
 2. a *tag* that uniquely identifies the block stored in the cache with respect to RAM
- As an instance, a 2048-entry cache, each 32-byte, contains overall $2048 \times 32 = 64\text{KB}$ of data (or instructions)

Direct-mapped cache

- In a direct-mapped cache, each RAM block is stored in a single cache location.
- So, given a memory address, there is a single precise position in the cache where to look for. If it is in the cache, it is there.
- To find the cache location containing the RAM block holding the addressed data or instruction, the computation is:

(block address in RAM) modulo (number of entries in the cache)

- The computation is straightforward if the number of cache entries is a power of 2

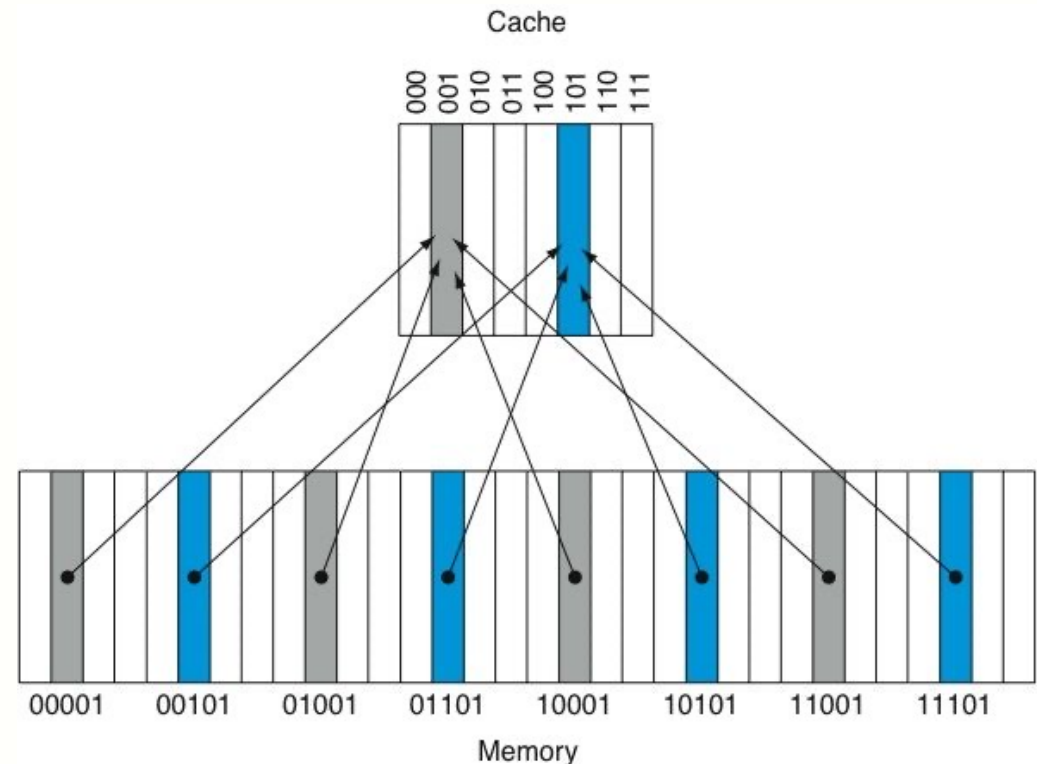
what is a “block address”?

Direct-mapped cache

- The PC, a LOAD or a STORE specify a word address to be accessed (an instruction addressed by the PC, or a data item for reading or writing).
- How can we get the RAM address of the block holding that peculiar word?
- Once obtained that address, how can we check if the block is already in the cache?
- And if the block is in the cache, how do we get the word on the basis of the address specified through the PC, LOAD or STORE?

Direct-mapped cache

- (Patterson-Hennessy, fig. 7.5).
As an example, let us assume a RAM split into 32 blocks one byte each, and an 8-entry cache (a byte each entry, of course).
- In which cache entry is stored the block with address 01001?



- $01001 \bmod 1000 = 001$. But note that cache entry 001 will match also blocks at addresses 00001, 10001, 11001.
- What discriminates all blocks that can be stored in cache entry 001? Obviously, the two most significant bits of the address of each block, *that are used as the tag associated to the cache line.*

Direct-mapped cache

- So, when the CPU requires block 01001, first of all it checks if cache entry 001 has its validity bit set (the entry could as well contain no data at all).
- If the bit is set, entry 001 tag is compared with the two most significant bits of the address of the requested block.
- If there is a match, cache entry 001 holds the requested line, otherwise this is a cache miss (the same happens if the validity bit is not set).

Direct-mapped cache

- In real cases, the RAM is organized in blocks with larger dimensions, and the CPU addresses a word (instruction or data) smaller than the block holding that word, but the scheme is the same.
- Let us assume 32-byte (2^5 byte) blocks, logically organized as 8 4-byte words and a 2048-entry (2^{11} entries) cache.
- To store/fetch data from the cache, the 32-bit address generated from the CPU is split into 4 sections. For a 2048-entry cache:

TAG (16 bit)	INDEX (11 bit)	WORD (3 bit)	BYTE (2 bit)
--------------	----------------	--------------	--------------

Direct-mapped cache

- Where:
 - **TAG**: the 16 most significant bits in the CPU generated address.
 - **INDEX**: the cache entry holding the data item, if present (note: $2^{11}=2048$)
 - **WORD**: which 4-byte word within the 32-byte block is actually referenced ($2^3 = 8$; $8 \times 4 = 32$)
 - **BYTE**: usually unused, it specifies the byte referenced within the word.
 - WORD and BYTE make up the **DISPLACEMENT**²³

Direct-mapped cache

- The block whose address in RAM is:

“TAG – INDEX – 0 0 0 0 0”

 16 bit 11 bit

- contains all RAM bytes from address


“TAG – INDEX – 0 0 0 0 0”

to address

“TAG – INDEX – 1 1 1 1 1”

- Which addresses have the 8 4-byte words stored in line with address “TAG – INDEX – 0 0 0 0 0”?

Direct-mapped cache

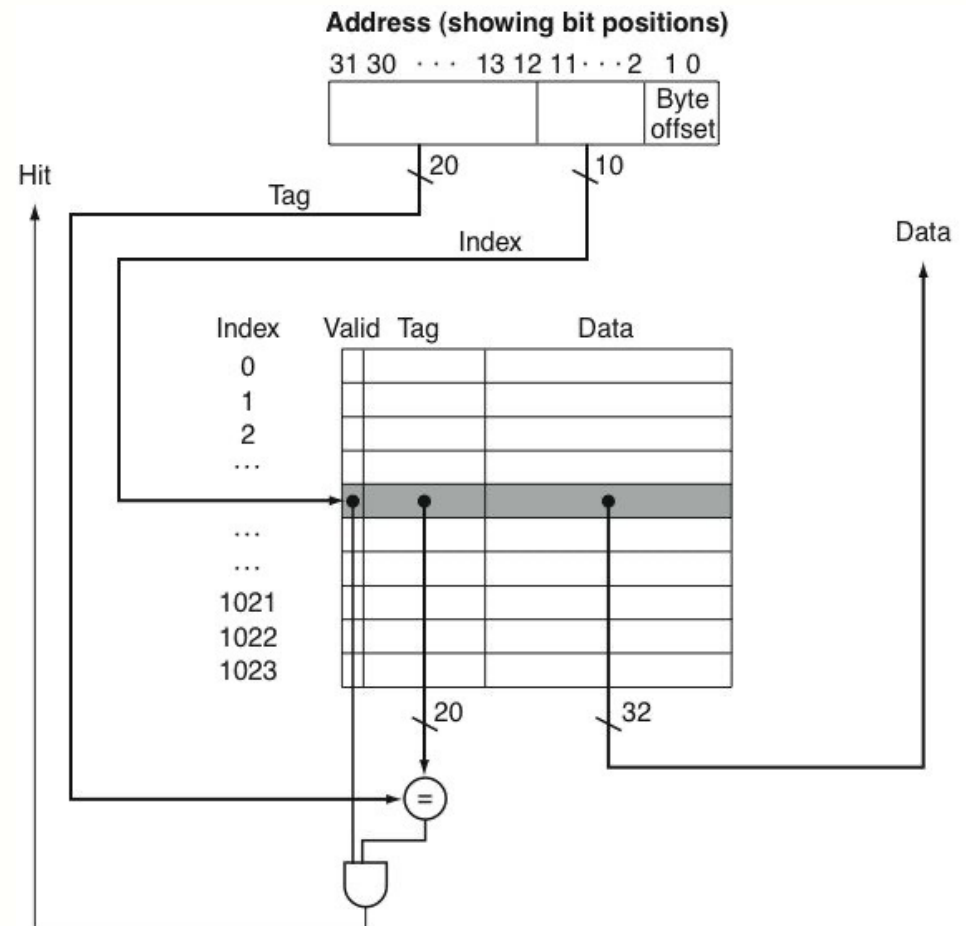
- When the CPU generates an address, the hardware extracts the 11 bits for INDEX, and uses them to address one of the 2048 entries in the cache
 - Question 1: why are the remaining 5 least significant address bits unused ?
 - Question 2: what about the computation:
(RAM block address) modulo (Number of cache entries)
what is the “block address”?
- If the corresponding entry is valid (as specified by the validity bit), the TAG field in the cache entry and the TAG address field are compared (the 16 most significant address bits).
- If they are equal, this is a **cache hit**. Through the WORD field, only the addressed word is extracted from the cache (the same happens with the BYTE field, if a specific byte is addressed).

Direct-mapped cache

- If the entry is invalid, or the two TAGs do not match, this is a **cache miss**.
- The missing block is fetched from RAM and it replaces the one (possibly) present (here is the *temporal locality* principle: an item used more recently substitutes an older one)
- Obviously, a cache miss always produces a waste in time (**slightly**) longer than the time required to directly fetch the data from RAM.
- In any case, all these operations are carried out in parallel in hardware, for maximum speed.

Direct-mapped cache

- A direct-mapped cache with 1024 4-byte entries. The 20 most significant bits are used as a tag, the 10 intermediate ones as block index, the remaining 2 least significant bits for addressing the byte within the block (Patterson-Hennessy, fig. 7.7)
- Should we address 2-byte words within the blocks, which part of the CPU generated address would we use?



Cache dimension

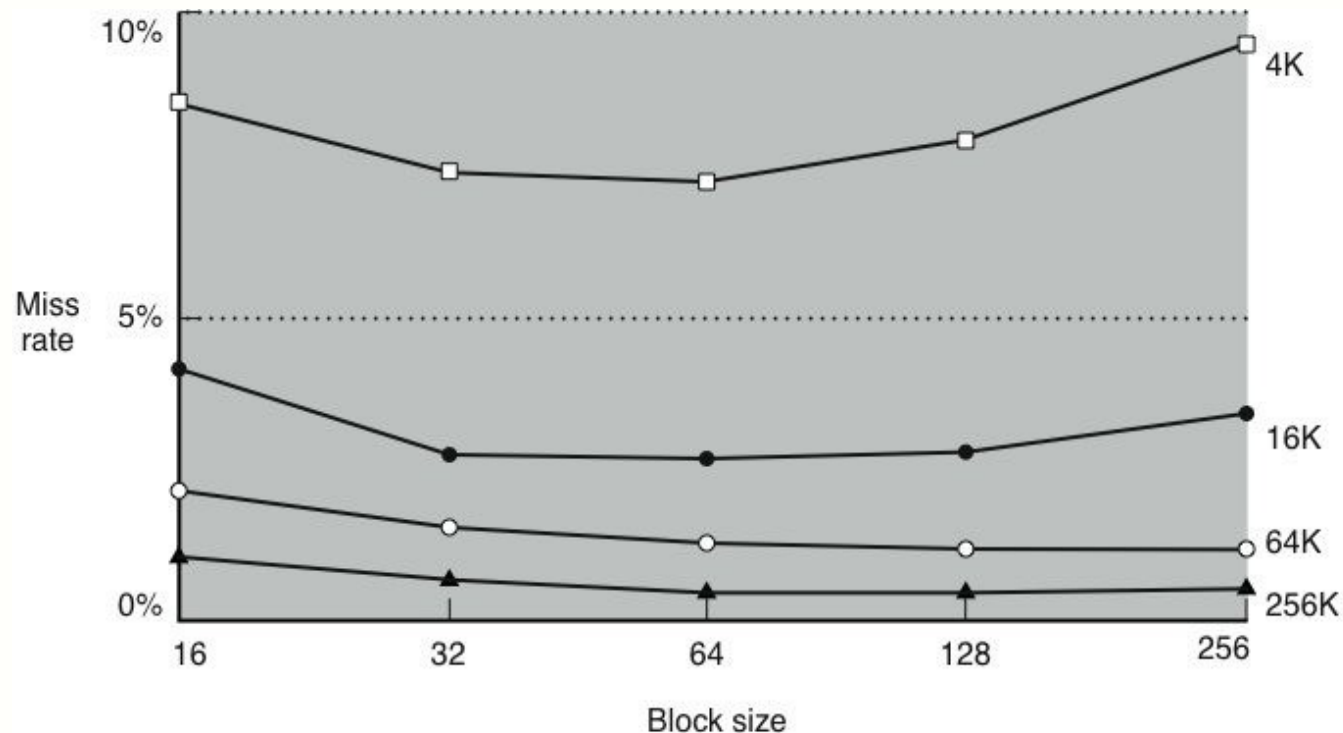
- The “effective” cache dimension is larger than the “nominal” one: for each block, it is necessary to store the tag block and its validity bit.
- Let us consider for example a 16 Kbyte cache (2^{14} bytes = 16384 bytes) with 1024 entries (2^{10}), so 16-byte blocks (2^4) byte.
- If the address is 32 bits, each tag is composed of $32 - 10 - 4 = 18$ bits, plus the validity bit, so that each entry has $128 + 19 = 147$ bits.
- Overall, the cache stores $1024 \cdot 147 = 150528$ bits, that is 18816 bytes.

Block dimension

- Cache performances depend also on block dimension. Generally speaking, larger blocks allow to reduce cache misses, because they exploit better spatial locality.
- However, block dimension cannot be enlarged beyond a certain level, because, for a given cache capacity, the larger the block size, the smaller the number of blocks in the cache, so the higher the probability to substitute a block (cache miss) before all of its data are actually used.

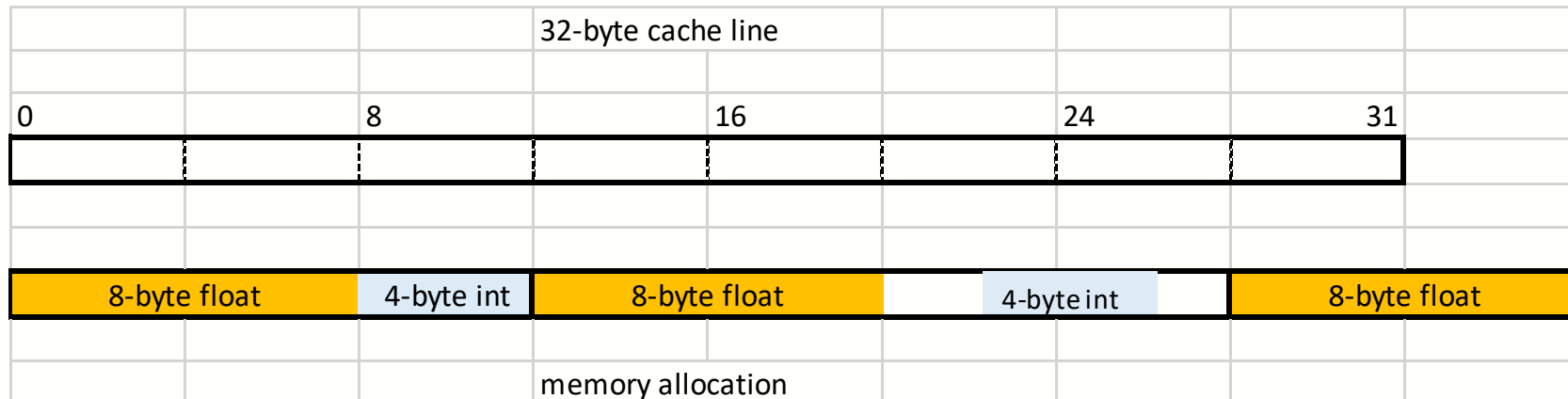
Block dimension

- The picture (Patterson-Hennessy, 7.8) depicts cache miss frequency vs block size and cache size in SPEC92 benchmark. Larger caches allow to use larger blocks (and still cause a smaller number of cache misses).



Address alignment & caches

- An address is *aligned* if it is congruent with the object it references
- Ex: a 32-bit integer variable INT is aligned if $\&INT \bmod 4 = 0$
- Objects can be *instructions* or *data* – RISC architectures are designed for instruction alignment, CICS exhibit extensive mis-alignment.
- Proper alignment takes into account *cache line length*



Cache miss management

- Causes for misses:
 - **Compulsory** misses
 - first reference to a block
 - **Capacity**
 - A block is discarded to make room for other blocks and later it is referenced again
 - 32K cache, scan of a 128K array
 - **Conflict**
 - A program references multiple addresses that map to the same block in the cache
 - Two 8K arrays that map into the same cache lines

Cache miss management

- What happens in case of cache miss? The pipeline must be stalled, until the missing instruction or data item is fetched from RAM.
- In case of miss in the I-cache, the current PC must be decremented by the same amount by which it was increased during the IF phase.
- When the block containing the missing instruction/data item has been loaded, execution proceeds. With a missing instruction, the IF phase is executed, with a missing data item, the MEM one.

Write policy

- Handling writes is more complex, since data must be changed in RAM as well, sooner or later.
- When a cache block receives a write, a first option is to propagate immediately the change to the RAM, thus maintaining **cache coherence**.
- In order to maintain cache coherence, with a cache miss on write the missing data item is fetched from RAM, modified in the cache, and written back to RAM.
- This write policy is called **write-through**: writes are always performed both in the cache and in RAM, so that the data present in both memories are always consistent.

Write policy

- Write-through exhibits poor performance, since it requires access to RAM for every write in the cache, an operation that takes a lot of clock cycles.
- A solution to this problem is using a **write buffer**: a small memory in the CPU that hosts a data item, until it is written to RAM.
- Once the CPU has written in the cache and in the write buffer, it goes on executing, and the write buffer takes care of forwarding the data to RAM. The buffer is released when the write operation is complete.

Write policy

- In true architectures, the write buffer holds more entries, to serve multiple frequent write requests: when the buffer gets filled (all data it holds still have to be moved to RAM), the pipeline must be stalled (unless the processor uses dynamic scheduling of the pipeline).

Write policy

- An alternative scheme is **write-back**, which only modifies data in the cache (it is sometimes called *write deferred*).
- The write is forwarded to RAM (or to lower cache levels, if there are multiple cache levels) only if the data item has to be replaced with another data item having the same tag.
- Write-back requires more data to be stored at each cache block: besides the validity bit, it is necessary to use a **dirty bit**, which, if set, signals that the block has been modified.

Write policy

- **Write-back** requires a policy to synchronize the cache with lower levels of the hierarchy and RAM in the end.
- A block with dirty bit set is written back in the rest of the hierarchy only when the block must be replaced.

Write policy

- **Write allocate** : the block is allocated first (brought to the cache) then it undergoes one of the write actions .
- **No-write allocate** : the cache is *not* affected and the block is written only in lower-level memory hierarchy. The updated block comes to cache only in a subsequent read reference.
- Usually write-back is paired with write allocate, while write-through obviously favors no-write allocate

RAM support

- The time required to fetch the data from RAM in case of a miss is called **miss penalty**.
- The length of miss penalty is due to the three operations necessary to handle a cache miss:
 1. Sending to RAM the address of the missing block
 2. Accessing the RAM to fetch the block
 3. Transmitting the block to the CPU

We disregard the cost of forwarding the block to the cache once it is within the CPU.

RAM support

- Miss penalty T_{miss}
 - T_{addr} time to send the ADDRESS of the block
 - T_{act} time to activate a memory row
 - T_{trans} time to transmit a memory word over the bus
- Given the length of a cache block and the structure of the memory, it is sufficient to establish the number of activations N_{act} and the number of transfers N_{trans} , so that the miss penalty is:
 - $T_{\text{miss}} = T_{\text{addr}} + N_{\text{act}} \times T_{\text{act}} + N_{\text{trans}} \times T_{\text{trans}}$
- This figure is usually expressed in number of EXTERNAL BUS clock cycles and then can be mapped to PROCESSOR's clock cycles₄₁

RAM support

- The first and the third operation can take a single clock cycle (of the bus clock, not the CPU's), while the second takes much longer, in the order of 10 to 20 clock cycles (of the bus, of course).
- Furthermore, a cache block is usually much larger than the amount of data that a memory bank can output with a single access.
- For instance, a block can have 16 bytes, while a RAM access allows to read a 32-bit word, that is 4 bytes, thus requiring 4 accesses to fetch a whole cache block.

RAM support

- Let us consider a 32-bit system (with a 32-bit bus).
Supposing that a RAM access takes 15 clock cycles (a reasonable assumption) to read a 32-bit word, to carry out a cache miss for a 16-byte block requires:
 - 1 clock cycle to send the missing block address to the RAM
 - 4×15 clock cycles to read the whole block
 - 4×1 clock cycle to transmit the whole block to the cache
- This gives a total of 65 clock cycles (bus clock cycles !).

RAM support

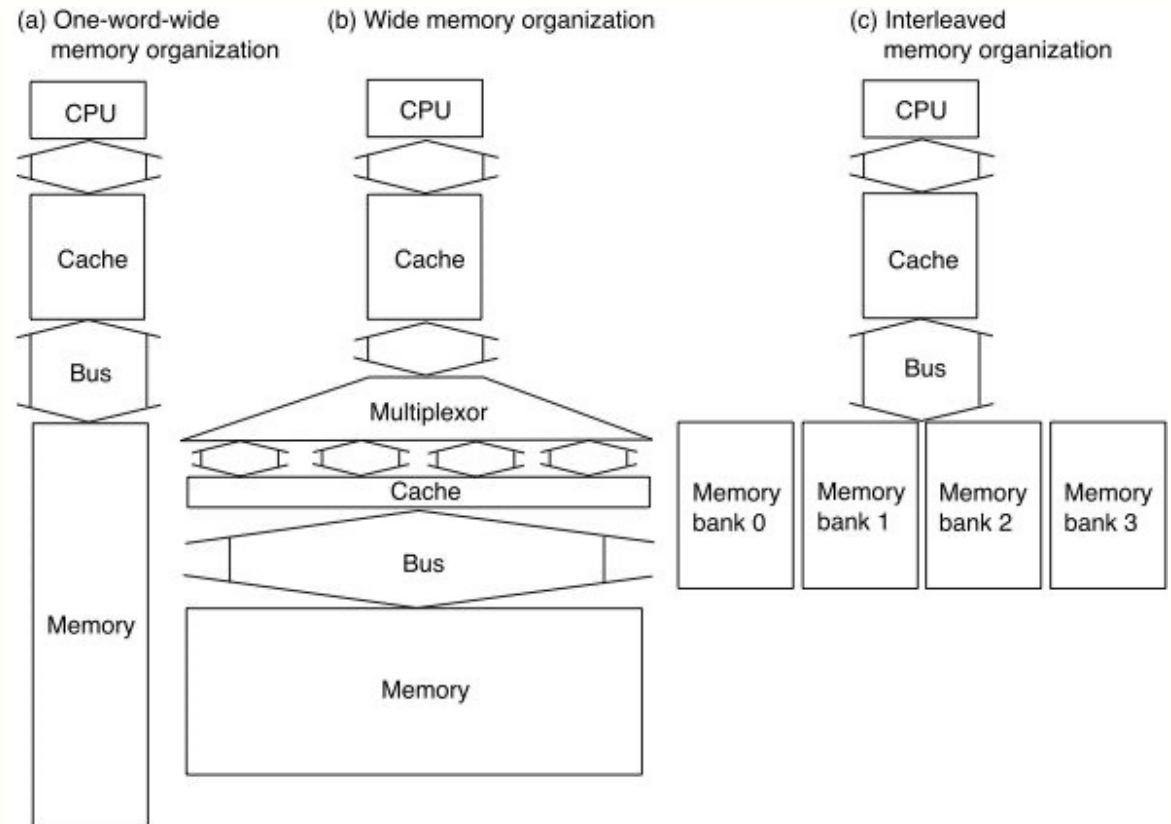
- Improvements can be obtained by:
 - increasing the bandwidth to the RAM, the amount of data that can be fetched from RAM in a single access (for instance, by having a whole block fetched in a single access).
 - widening the bus, so that the whole block extracted from RAM is transmitted to the processor in a single bus clock cycle.
- Widening the bus up to 16×8 bits in parallel is a very costly and difficult approach. All in all, even the increase in RAM bandwidth can alone yield considerable advantages.

RAM support

- To increase RAM bandwidth, it is necessary to set up the RAM with more banks, so that multiple words (not just a single one) can be read or written in a single clock cycle.
- Each bank gives access to a single word of a given block, and to read the (4, in our example) words in parallel, it suffices to send in parallel to each bank the address of block address they belong to.
- This scheme is called **interleaved memory** and allows to spare on the time required to fetch a whole block from RAM.

RAM support

- Hennessy-Patterson, Fig 5.27: interlaced memory (c) allows to keep a manageable bus width and to speed up RAM access in cache miss.



- On the same case just considered, with approach (c) we get:
- 1 clock cycle to send the missing block address, 15 clock cycles to get the 4 words of the block (the 4 banks are accessed in parallel), 4×1 clock cycles to transmit the whole to the cache, with a grand total of 20 clock cycles, against 65 for case (a)

RAM support

- Miss penalty $T_{\text{miss}} = T_{\text{addr}} + N_{\text{act}} \times T_{\text{act}} + N_{\text{trans}} \times T_{\text{trans}}$
taking into account **cache structure**, memory **interleaving** and system **bus width** (using memory chips supporting DDR technology)
- LB - Length of a cache Block
- IF - Interleaving Factor
- MW - Memory bank Width
- BW - system Bus Width
- $N_{\text{act}} = \text{LB} / (\text{IF} \times \text{MW})$
- $N_{\text{trans}} = \text{LB} / \text{BW}$

Memory Technology

- Performance metrics
 - *Latency* is concern of cache
 - *Bandwidth* is concern of multiprocessors and I/O
 - *Access time*
 - Time between read request and when desired word arrives
 - *Cycle time*
 - Minimum time between unrelated requests to memory
- DRAM used for main memory, SRAM used for cache

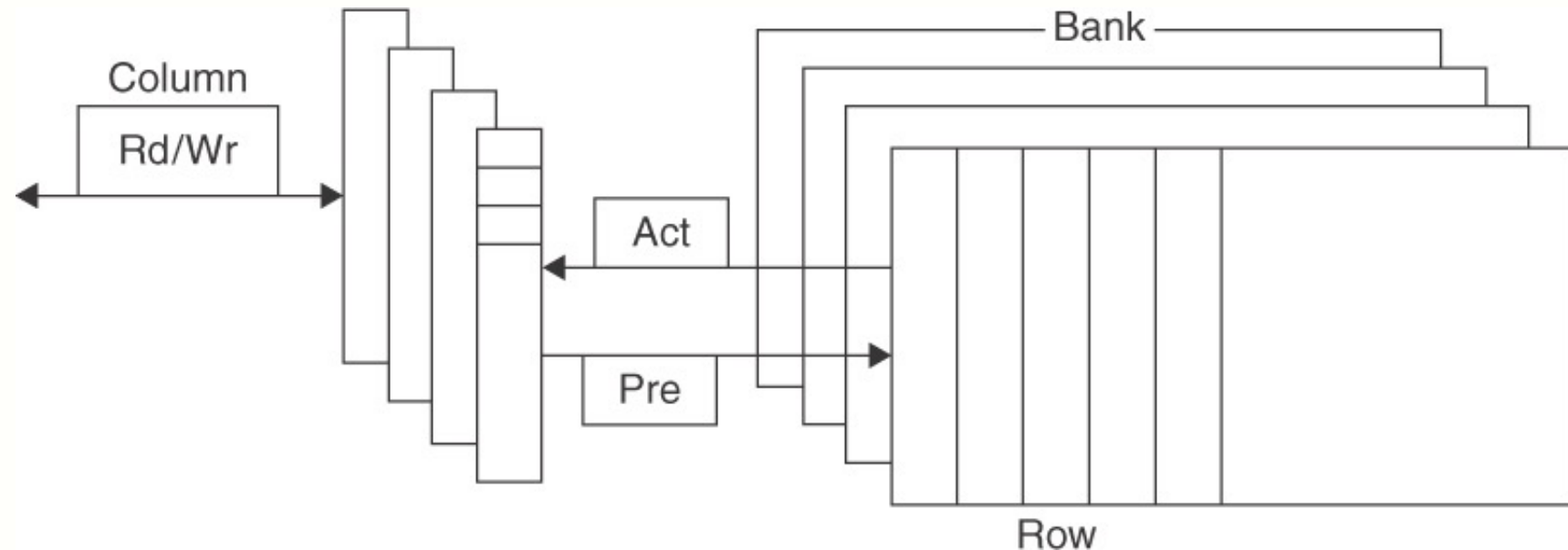
Memory Technology

- SRAM
 - Requires low power to retain bit
 - Requires 6 transistors/bit
- DRAM
 - Must be re-written after being read
 - Must also be periodically refreshed
 - Every ~ 8 ms
 - Each row can be refreshed simultaneously
 - One transistor/bit
 - Address lines are multiplexed:
 - Upper half of address: row access strobe (RAS)
 - Lower half of address: column access strobe (CAS)

Memory Technology

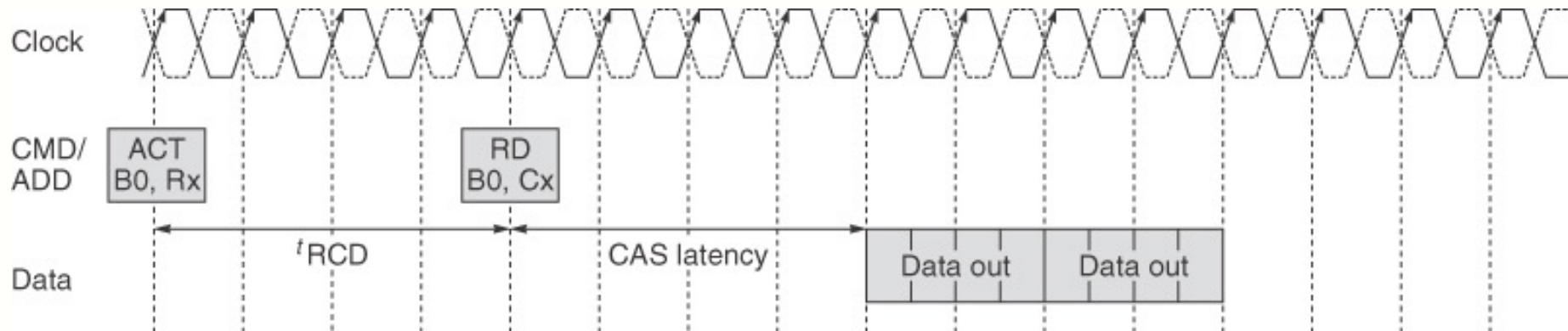
- Amdahl:
 - Memory capacity should grow linearly with processor speed
 - Unfortunately, memory capacity and speed has not kept pace with processors
- Some optimizations:
 - Multiple accesses to same row
 - Synchronous DRAM
 - Added clock to DRAM interface
 - Burst mode with **critical word** first
 - Wider interfaces
 - Double data rate (DDR)
 - Multiple banks on each DRAM device

Memory Technology



HP5- Fig. 2.12 Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows. Sending a PRE (precharge) command opens or closes a bank. A row address **RAS** is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses **CAS** at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address. Each command, as well as block transfers, are synchronized with a clock.

Memory Technology



HP5 - Figure 2.31 DDR2 SDRAM timing diagram.

Memory Technology

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

Figure 2.13 Times of fast and slow DRAMs vary with each generation. (Cycle time is defined on page 95.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs. The introduction of various burst transfer modes in the mid-1990s and SDRAMs in the late 1990s has significantly complicated the calculation of access time for blocks of data; we discuss this later in this section when we talk about SDRAM access time and power. The DDR4 designs are due for introduction in mid- to late 2012. We discuss these various forms of DRAMs in the next few pages.

Memory Technology

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /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	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

Figure 2.14 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2010. Note the numerical relationship between the columns. The third column is twice the second, and the fourth uses the number from the third column in the name of the DRAM chip. The fifth column is eight times the third column, and a rounded version of this number is used in the name of the DIMM. Although not shown in this figure, DDRs also specify latency in clock cycles as four numbers, which are specified by the DDR standard. For example, DDR3-2000 CL 9 has latencies of 9-9-9-28. What does this mean? With a 1 ns clock (clock cycle is one-half the transfer rate), this indicate 9 ns for row to columns address (RAS time), 9 ns for column access to data (CAS time), and a minimum read time of 28 ns. Closing the row takes 9 ns for precharge but happens only when the reads from that row are finished. In burst mode, transfers occur on every clock on both edges, when the first RAS and CAS times have elapsed. Furthermore, the precharge is not needed until the entire row is read. DDR4 will be produced in 2012 and is expected to reach clock rates of 1600 MHz in 2014, when DDR5 is expected to take over. The exercises explore these details further.

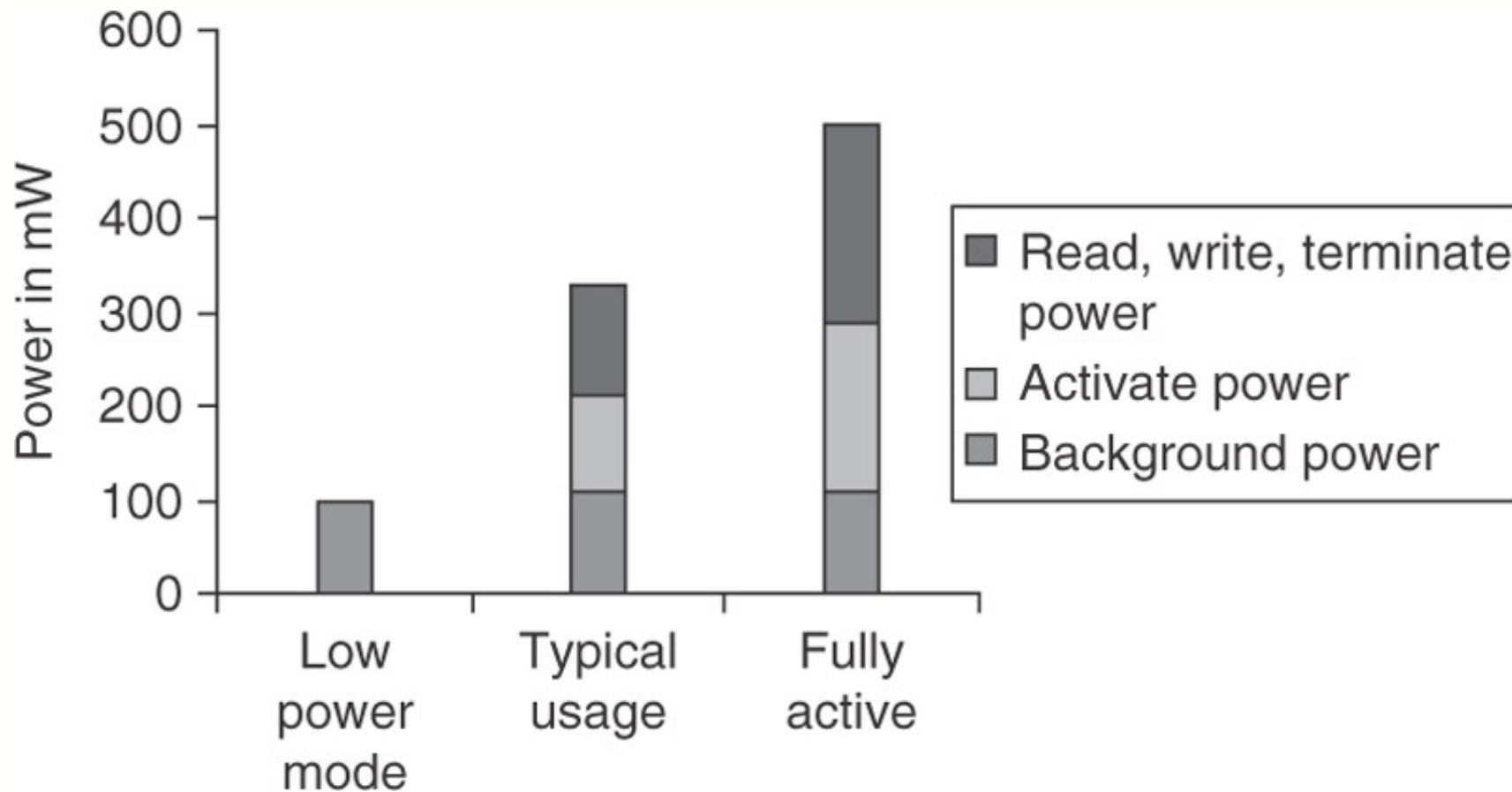
Memory Technology

Standard name	Memory clock (MHz)	I/O bus clock (MHz)	Data rate (MT/s) ^[c]	Module name	Peak transfer rate (MB/s) ^[d]	Timings CL-tRCD-tRP	CAS latency (ns)
DDR4-1600J*	200	800	1600	PC4-12800	12800	10-10-10	12.5
DDR4-1600K						11-11-11	13.75
DDR4-1600L						12-12-12	15
DDR4-1866L*	233.33	933.33	1866.67	PC4-14900	14933.33	12-12-12	12.857
DDR4-1866M						13-13-13	13.929
DDR4-1866N						14-14-14	15
DDR4-2133N*	266.67	1066.67	2133.33	PC4-17000	17066.67	14-14-14	13.125
DDR4-2133P						15-15-15	14.063
DDR4-2133R						16-16-16	15
DDR4-2400P*	300	1200	2400	PC4-19200	19200	15-15-15	12.5
DDR4-2400R						16-16-16	13.32
DDR4-2400T						17-17-17	14.16
DDR4-2400U						18-18-18	15
DDR4-2666T	333.33	1333.33	2666.67	PC4-21300	21333.33	17-17-17	12.75
DDR4-2666U						18-18-18	13.50
DDR4-2666V						19-19-19	14.25
DDR4-2666W						20-20-20	15
DDR4-2933V	366.67	1466.67	2933.33	PC4-23466	23466.67	19-19-19	12.96
DDR4-2933W						20-20-20	13.64
DDR4-2933Y						21-21-21	14.32
DDR4-2933AA						22-22-22	15
DDR4-3200W	400	1600	3200	PC4-25600	25600	20-20-20	12.5
DDR4-3200AA						22-22-22	13.75
DDR4-3200AC						24-24-24	15

Memory Technology

- DDR:
 - DDR2
 - Lower power (2.5 V -> 1.8 V)
 - Higher clock rates (266 MHz, 333 MHz, 400 MHz)
 - DDR3
 - 1.5 V
 - 800 MHz
 - DDR4
 - 1-1.2 V
 - 1600 MHz
- GDDR5 is graphics memory based on DDR3

Memory Technology



HP5 - Figure 2.15 Power consumption for a 2 Gb DDR3 SDRAM operating under three conditions: low power (shutdown) mode, typical system mode (DRAM is active 30% of the time for reads and 15% for writes), and fully active mode, where the DRAM is continuously reading or writing when not in precharge. Reads and writes assume bursts of 8 transfers. These data are based on a Micron 1.5V 2Gb DDR3-1066.

Set-Associative caches

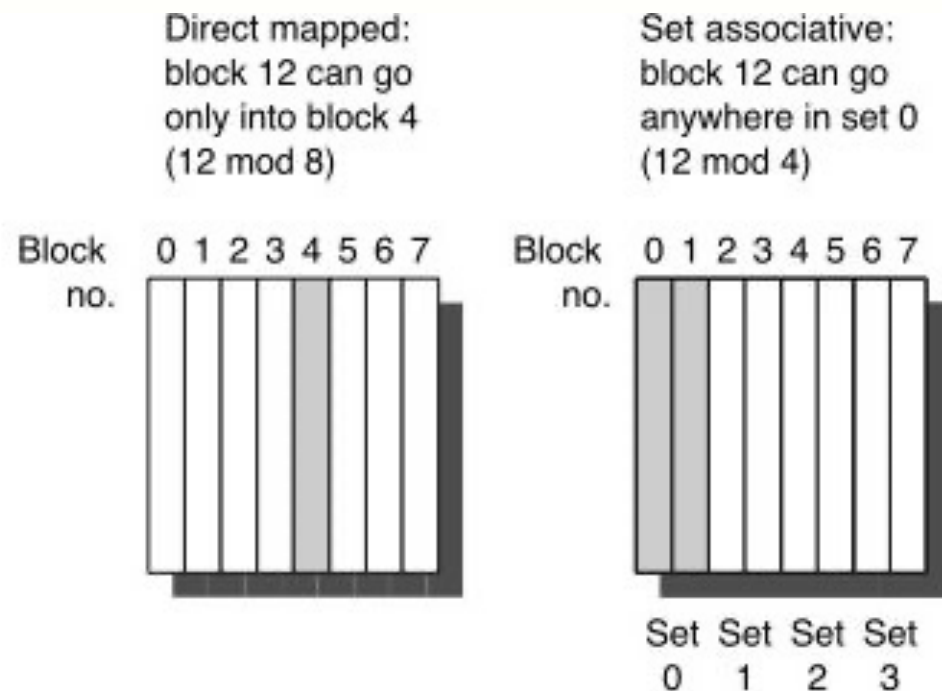
- A basic feature of direct-mapped caches is that a RAM block is always stored in the same cache entry.
- So, multiple RAM blocks match the same cache entry, since the cache is much smaller than the RAM.
- Allowing a more flexible block placement in the cache, it is possible to reduce cache misses, thus increasing system performance.
- This result can be obtained with **n-way set-associative caches**, where n is usually 2, 4 or 8.

Set-Associative caches

- A n-way set-associative cache is split into more sets, each containing n blocks.
- Each RAM block matches exactly a single set in the cache, and it can be stored in any of entries in the set.
- So, in a set-associative, the set (no longer the entry, as in a direct-mapped cache) containing a block is identified as:
 $(\text{RAM block address}) \bmod (\text{number of sets in the cache})$

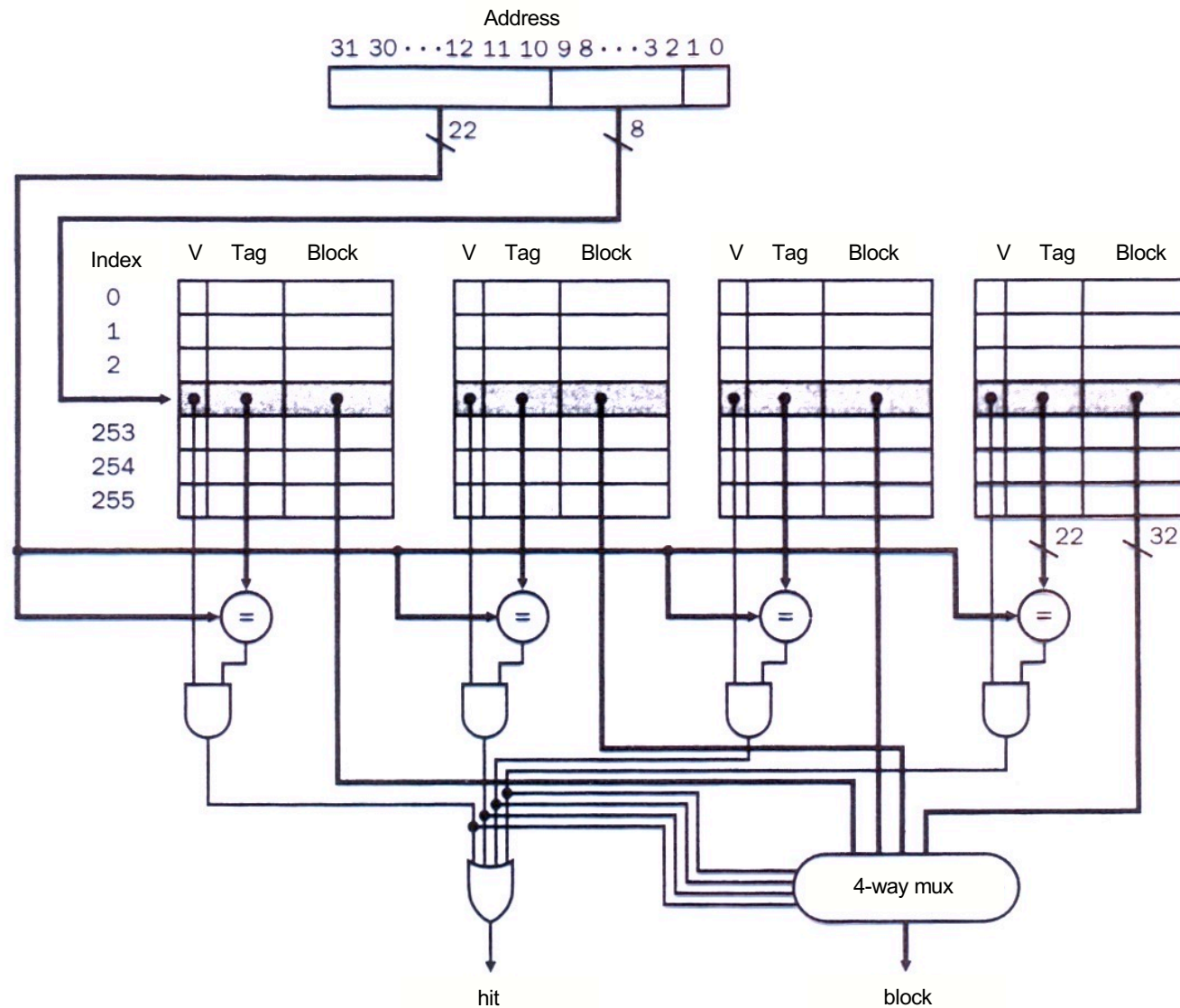
Set-Associative caches

- A direct mapped cache and a 2-way set-associative cache, both with a capacity of 8 blocks. In the first, block 12 can be placed only in entry number 4. In the second, the same block can be placed either in entry 0 or in entry 1, the are part of set “zero” (Hennessy-Patterson, Fig. 5.4)



Set-Associative caches

- A 4-way set-associative cache: 256 sets (8-bit index) each holding 4 blocks



Set-Associative caches

- Once identified the set containing a given block, it is still necessary to locate the entry holding the addressed block (through an associative search on all elements in the set).
- If the addressed block is missing, it must be brought into the cache, and if its set is full, one of the blocks of the set must be overwritten. Usually the “victim” block is the least recently used one (LRU block replacement policy).
- The circuitry required for this type of cache is more complex, but performance are better than with direct-mapped caches.

Set-Associative caches

- As an example, let us consider two caches, each 4-entry, the first is direct-mapped, the second is 2-way set-associative.
- Let us assume accesses to RAM blocks with addresses 0, 8, 0, 6, 8.
- How many cache misses do result?

Set-Associative caches

- For the direct-mapped cache, addressed blocks will be placed in the following cache entries:

Block address	Cache entry nella cache
0	$0 \text{ modulo } 4 = 0$
6	$6 \text{ modulo } 4 = 2$
8	$8 \text{ modulo } 4 = 0$

- References “0, 8, 0, 6, 8” cause 5 cache misses, since blocks 0 and 8 alternate within the same entry, and the first reference to any block causes a miss, obviously.

Cache Set-Associative

- For the 2-way set-associative cache, the 4 entries are split into 2 sets, indexed as 0 and 1, and addressed blocks will be placed in the sets as follows:

Block address	Set
0	$0 \text{ modulo } 2 = 0$
6	$6 \text{ modulo } 2 = 0$
8	$8 \text{ modulo } 2 = 0$

- With references “0, 8, 0, 6, 8”, blocks 0 and 8 are placed in separate entries of the same set, so that the second reference to block 0 does not generate a cache miss. When block 6 is addressed, (causing a cache miss), block 8 is removed from the cache (LRU policy), and the subsequent reference to block 8 generates a cache miss. Overall, there are **4** cache misses.

Set-Associative caches

- If the set-associative cache were 4-way, one can easily check that the previous sequence would result in 3 cache misses.
- By extension, an associative cache with a *single*, large set, where a block can be placed in any entry, is called **fully associative cache**.
- This type of cache yields best performance as to cache misses, but caches with (a few) thousands entries are much too complex and expensive to build.
- Still, the principle is valid: the larger the number of entries available for placing blocks (that is, the larger n , the number of ways), the better the performances.

Set-Associative caches

- Miss rate in a D-cache for different types of caches with the same capacity (using SPEC2000 as benchmark) (Patterson-Hennessy, fig. 7.15)

Associativity	Miss rate
1-way (direct mapped)	10.3%
2-way	8.6%
4-way	8.3%
8-way	8.1%

- The advantages with different associativity seem minimal, but they are not, considering the cost of serving even a single miss...

Performance: a simple example

- What is the effect of the cache subsystem on CPU performance ? Let us make a simple estimate (with reasonable values):
 - instruction execution time (CPI): 1 clock cycle
 - average miss rate: 2%
 - average memory reference per instruction: 0,2
 - time to fetch a data item from memory in case of miss: 100 clock cycles (CPU)

Performance: a simple example

- What is the CPI, assuming a “perfect” cache? (cache miss = 0, access time = 0). Answer = 1 CPI
- What would be the CPI, with no cache at all?
 - $\text{CPI} = 1 + 100 \times 0,2 = 21$
- And with a “real” cache?
 - $\text{CPI} = 1 + 0,02 \times 100 \times 0,2 = 1,4$ (yet, assuming a cache with 0 access time)

Performance

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction count}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

Average memory access time = Hit time + Miss rate \times Miss penalty

$$\begin{aligned} \text{AverageMemoryAccessTimeHierarchy} &= \\ &= \text{HitTime}_{L1} + \text{MissRate}_{L1} \times (\text{HitTime}_{L2} + \text{MissRate}_{L2} \times \text{MissPenalty}_{L2}) \end{aligned}$$

Performance

$$\text{CPUtime} = \text{IC} \times \text{CPI} \times \text{ClockCycleTime}$$

$$\text{CPI} = \text{CPI ideal} + \text{averageClockCyclesLostStall}$$

Let us consider only the effect of the memory subsystem only

$$\text{averageClockCyclesLostStall} = A + B + C \text{ where}$$

$$A = \text{CyclesLostRead}$$

$$B = \text{CyclesLostWrite}$$

$$C = \text{CyclesLostFetch}$$

$$A = \%Read \times \text{MissRateonRead} \times \text{MissPenaltyonRead}$$

$$B = \%Write \times \text{MissRateonWrite} \times \text{MissPenaltyonWrite}$$

$$C = \text{MissRateonFetch} \times \text{MissPenaltyonFetch}$$

I-cache for Fetch D-cache for Read and Write

A and B may differ, because of write policy

in WriteThrough $\text{MissPenaltyonRead} = \text{MisspenaltyonWrite}$

in WriteBack another model is required

Let us assume WriteThrough and a simplified behaviour

$$A = B = (\%Read + \%Write) \times \text{MissRate} \times \text{MissPenalty}$$

$$\text{CPI real} = \text{CPI ideal} +$$

$$\text{MissRateonFetch} \times \text{MissPenaltyonFetch} +$$
$$(\%Read + \%Write) \times \text{MissRate} \times \text{MissPenalty}$$

Performance

Example

CPI ideal = 1,5

From benchmark

From memory subsystem

From profiling

%Read+%Write = 36%

MissPenalty = 200

MissRate: Icache = 2%, Dcache = 4%

CycleLostInstructions

$2\% \times IC \times 200$

CycleLostDataAccess

$36\% \times IC \times 4\% \times 200$

TotalCycleLost

$(4 + 2,88) \times IC$

averageClockCyclesLostStall

$(4 + 2,88) = 6,88$

CPI real = CPI ideal + averageClockCyclesLostStall

$= 1,5 \quad + \quad 6,88$

$= 8,38$

Performance

Enhancements

- 1) “Ideal memory” : no memory stalls
- 2) “Ideal architecture”: no stalls due to conflicts
- 3) “Improve CPU speed only”: overclocking
- 4) “Improve memory subsystem”: reducing miss cost

Performance

1) “Ideal memory” : no memory stalls

$$\frac{CPU\text{Time}\text{with}\text{Memory}\text{Stall}}{CPU\text{Time}\text{Ideal}} = \frac{CPI\text{ real}}{CPI\text{ ideal}} = \frac{1,5 + 6,88}{1,5} \cong 5,58$$

2) “Ideal architecture” : no stalls due to conflicts

a. CPI ideal = 1

b. No enhancement to memory subsystem, so memory stalls unchanged

$$\frac{CPU\text{Time}\text{with}\text{Memory}\text{Stall}}{CPU\text{Time}\text{Ideal}} = \frac{CPI\text{ real}}{CPI\text{ ideal}} = \frac{1 + 6,88}{1} \cong 7,88$$

Percentage of time lost due to memory stalls

a) Real case $\frac{6,88}{1,5+6,88} = 82\%$

b) “Ideal architecture” $\frac{6,88}{1+6,88} = 87\%$

Performance

3) “Improve CPU speed only”: **overclocking**

Rather than optimizing the architecture, lets us increase the speed of the Cpu clock

The memory subsystem remains untouched, so that the MissPenalty is constant in time, but its equivalence in CPUClockCycles changes

Assumption: clock frequency **doubled**

MissPenalty (in CPU clock cycles)	= 2 * 200 = 400
CycleLostInstructions	2% x IC x 400
CycleLostDataAccess	36% x IC x 4% x 400
TotalCycleLost	(8 + 5.76) x IC
<i>averageClockCyclesLostStall</i>	(8 + 5,76) = 13,76

$$\begin{aligned}CPI_{boosted} &= CPI_{ideal} + averageClockCyclesLostStall \\ &= 1,5 + 13,76 \\ &= 15,26\end{aligned}$$

Speedup of overclocking

$$\frac{CPU_{TimeClock}}{CPU_{TimeClockBoosted}} = \frac{IC \times CPI \times ClockCycle}{IC \times CPI_{boosted} \times ClockCycleBoosted} = \frac{8,38}{15,26 \times 1/2} = 1,098$$

Performance

4) Improve memory subsystem": reducing miss cost

Let us assume that, by improving the RAM - bus - busclock organization the cost of the miss is reduced to half, so that

Assumption: MissPenalty = 100

The processor clock and its architecture are untouched

MissPenalty (in CPU clock cycles) = 100
CycleLostInstructions 2% x IC x 100
CycleLostDataAccess 36% x IC x 4% x 100
TotalCycleLost (2 + 1,44) x IC
averageClockCyclesLostStall (2 + 1,44) = 3,44

$CPI_{missreduced} = CPI_{ideal} + averageClockCyclesLostStall$
 $= 1,5 + 3,44$
 $= 4,94$

Speedup versus **overclocking** and vs **standard configuration**

$$\frac{CPU_{TimeClockBoosted}}{CPU_{TimeMissReduced}} = \frac{IC \times CPI_{boosted} \times ClockCycleBoosted}{IC \times CPI_{missreduced} \times ClockCycle} = \frac{15,26 \times \frac{1}{2}}{4,94} = 1,56$$

$$\frac{CPU_{TimeClock}}{CPU_{TimeMissReduced}} = \frac{IC \times CPI \times ClockCycle}{IC \times CPI_{missreduced} \times ClockCycle} = \frac{8,38}{4,94} = 1,69$$

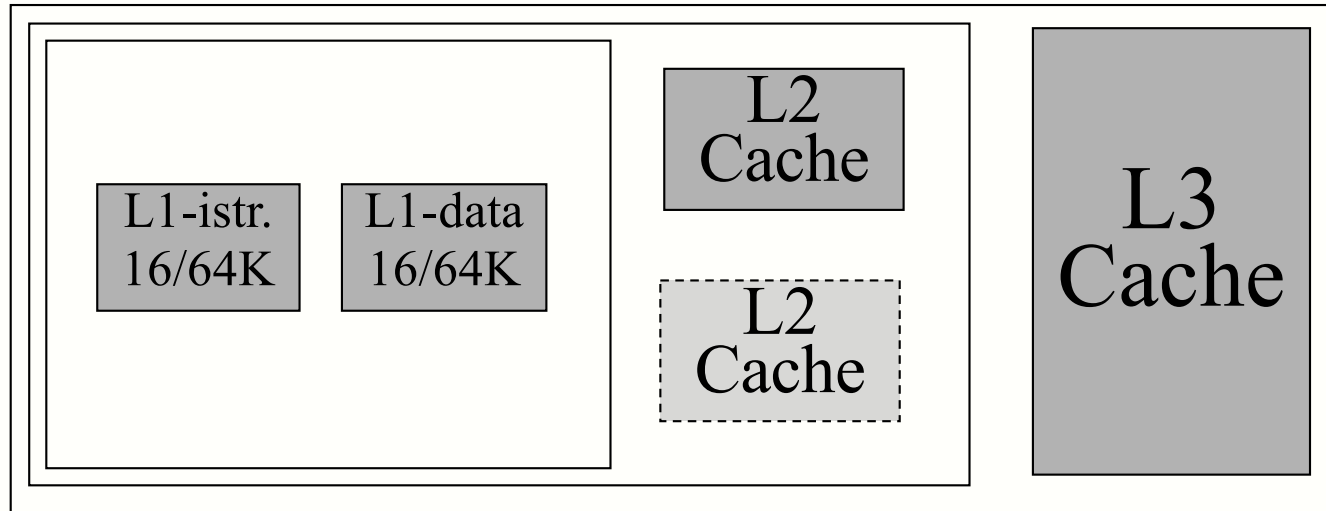
Increase in performance

- A cache has better effects on system performances :
 1. the smaller (in time) the hit time
 2. the smaller (in time) the cost of a miss
 3. the smaller the miss rate

Reducing the cost of a cache miss

- The performance gap between RAM and CPU widens in time, so the cost of a miss gets higher and higher.
- Memory interleaving is a technique to reduce the time required to fetch from RAM a missing block.
- Another way to reduce cache miss cost consists of using **multiple levels of cache**, to avoid going down to main memory to get the missing block.
- With this approach, it is possible to find a reasonable trade-off between the requirements of having a cache running at the same speed as the CPU (expensive, and so small), and a cache large enough to hold a sufficient fraction of RAM (slower, but less expensive)

Multilevel caches



- In a multilevel cache system caches are usually inclusive, that is :
 - $L1 \subseteq L2 \subseteq L3$
- In modern multi-core processors, each core has private L1 and L2 caches, while L3 is shared. L1 and L2 are on the same chip as the processor, while L3 can be: on the same chip; on a separate chip in the same package; on the motherboard.

Multilevel caches

- In a system with at least two cache levels, the following should be true:
 1. first level caches have an access time close to or equal to CPU speed, thus causing no delay in case of cache hit.
 2. the second level cache is sufficiently larger than first level ones, so that the number of second level misses is considerably smaller than first level misses.
 3. the third level cache has the same relationship with the second level one.

Multilevel caches

- Current processors exhibit the following figures in caches:

Cache	latency	dimension
L1	1 ÷ 4 clock cycles	16 ÷ 64 Kbytes
L2	10 ÷ 15 clock cycles	256 ÷ 1024 Kbytes
L3	40 ÷ 50 clock cycles	2 ÷ 10 Mbytes

Multilevel caches

- The three-level cache hierarchy of Intel i7:

Cache	access time	associativity	dimension
L1 D-cache	4 cycles	8-way	32 Kbytes
L1 I-cache	4 cycles	4-way	32 Kbytes
L2	10 cycles	8-way	256 Kbytes
L3	35 cycles	16-way	2 Mbytes per core

Performance 2-level caching

$$\begin{aligned} \text{AverageMemoryAccessTimeHierarchy} &= \\ &= \text{HitTime}_{L1} + \text{MissRate}_{L1} \times (\text{HitTime}_{L2} + \text{MissRate}_{L2} \times \text{MissPenalty}_{L2}) \end{aligned}$$

A. One level only: L1

$$\begin{aligned} \text{CPI} &= \text{CPI}_{\text{exe}} + \text{HitRate}_{L1} \times \text{HitPenalty}_{L1} + \text{MissRate}_{L1} \times (\text{HitPenalty}_{L1} + \text{MissPenalty}_{\text{RAM}}) \\ &= \text{CPI}_{\text{exe}} + \text{Latency}_{L1} + \text{MissRate}_{L1} \times \text{MissPenalty}_{\text{RAM}} \end{aligned}$$

B. Two levels: L1_B – L2

Usually the L1_B cache has a different dimension and a different latency with respect to L1

$$\text{CPI} = \text{CPI}_{\text{exe}} + \text{Latency}_{L1B} + \text{MissRate}_{L1B} \times (\text{Latency}_{L2} + \text{Missrate}_{L2} \times \text{MissPenalty}_{\text{RAM}})$$

Performance 2-level caching

$$\begin{aligned} \text{AverageMemoryAccessTimeHierarchy} &= \\ &= \text{HitTime}_{L1} + \text{MissRate}_{L1} \times (\text{HitTime}_{L2} + \text{MissRate}_{L2} \times \text{MissPenalty}_{L2}) \end{aligned}$$

CPI exe	1,5							CPI	
One level	L1 latency	L1 Miss rate							
	10	4%						15,1	15,1
Two levels	L1 latency	L1 Miss rate	L2 Latency	L2 miss rate					
	4	20%	11	4%				8,42	8,42
Three levels	L1 latency	L1 Miss rate	L2 Latency	L2 miss rate	L3 Latency	L3 miss rate			
	4	20%	8	10%	25	4%		7,672	7,672
Penalty Ram	90								

Reducing cache misses

- It is the second best way to improve cache efficiency. There are multiple options:
 - First of all, set-associativity lowers cache miss rate.
 - A proper block dimension (with respect to cache capacity) optimizes cache misses for a given cache dimension.
 - Obviously, cache misses are reduced by enlarging the cache.

Reducing cache misses

- Cache dimensions (L2 and L3 especially) have increased steadily: in 2001 a L2 cache was as large as the main memory in a desktop system of 1991.
- If the cache is too small at any level with respect to the amount of data/instructions required by a program, there arises a true *thrashing* effect, similar to what happens with virtual memory (see the notion studied in Operating Systems)