Introduction

mercoledì 01 marzo 2017 11:49

Scalability

Not always it's possible to improve scalability by adding just hardware, because of the "system bottlenecks", like the shared memory in a CPU.
Bottleneck can only be avoided by changing the design during the development of the application.

Parallelism

- Data-level parallelism DLP
- Task-level parallelism TLP

Classes of architectural parallelism

Implicit parallelism	Explicit parallelism
Done by the compiler and by the hardware.	Explicitely made by the progammer.
• ILP: istruction-level parallelism ○ Pipeline ○ Speculative execution It works by pre-loading both "then" and "else" instructions, for example, and then throwing away the set that is not used.	• Vector architectures, for GPUs • Thread-level parallelism ○ Management of parallel thread Threads share code and memory (a shared one, for communications). This solution can be found in a server, in order to archive performances goale. • Request-level parallelism ○ Management of tasks In a web server, when the requests rise, the application can require more processes (like apache).

It's important to avoid condition jams (?) (w
condition that has still to be verified).
Solutions:
- Speculative execution
- Branch predictor
  It uses a branch table containing all th
  of code) of all possible jams.

Flynn's taxonomy

- SISD (single instruction stream, single data stream)
  Uniprocessors, like microcontrollers.
- SIMD
  - Vector architectures
  - Multimedia extensions
  - GPUs
- ~~MISD~~
- MIMD
  Schedulers use a shared memory, that acts as a bottleneck.
  The communication overhead between processes has to be minimized.
  - Tightly-coupled MIMD
  - Loosely-coupled MIMD

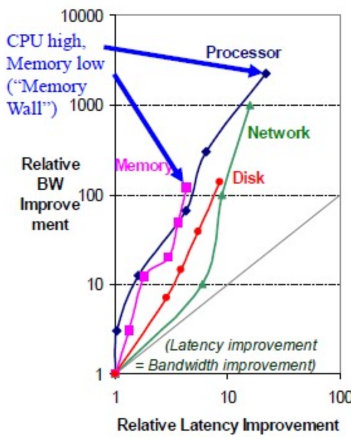
How to select the best architecture type for an application

- Desktop computing
  Minimum execution time between different architectures.
- Web server
  Best number of paged server per second (highest throughput)
- Cluster
  Number of pages for 1 Watt of power.

There could be different perfomance goals though.
For example a browser wants to minimize the latency between the request and the download of a page.
The webserver admin though wants to minimize the cost of running the server, so the main goal is to have a better a number of pages / Watts rate.

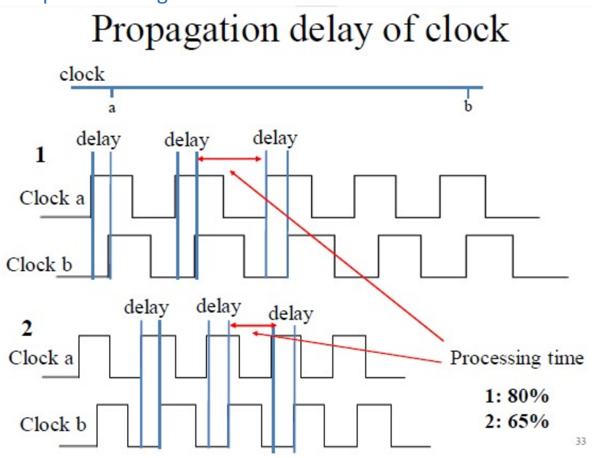
Memory latency

That affects the application efficiency because the CPU has to fetch instructions from the memory.
Multi-level cache can reduce this latency.
The memory is the main reason we can't speed up a computer system as the following graph shows:



So, during a development of an applications, an engineer should minimize the memory usage.

Increasing the clock speed: is it a good idea?



The delay is due to the capacitance of the bus wire.
The logic can only start when the value on the bus is stable.

Cons of increasing the clock speed

1. High speed components consumes a lot of power.

Performance evaluation

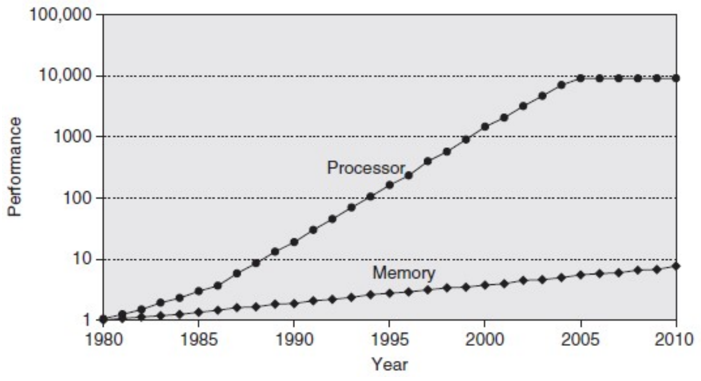
$$CPU\ time = \frac{seconds}{program} = CPU\ clock\ cycles\ for\ a\ program \times Clock\ cycle\ time =$$

$$= Instruction\ count \times Cycles\ per\ instruction \times Clock\ cycle\ time$$

Cache

martedì 16 maggio 2017 09:25

- Why
    - To let programmers have unlimited amounts of **fast** memory.
    - CPU memory requests per seconds wins over DRAM accesses per second, as shown below:



- It exploits:
    - Principles of locality
        - **Temporal locality:** recently accessed items (data and instructions) are likely to be accessed in the near future.
            - Programs tend to reuse data and instructions they have used recently.
            - "A program spends 90% of its execution time in only 10% of the code".
        - **Spatial locality**: items whose addresses are near one another tend to be referenced close together in time.
    - Trade-off in the cost-performance of memory technologies
- Objective: optimizing average memory access time
    - Cache access time
    - Miss rate
    - Miss penalty

Cache blocks

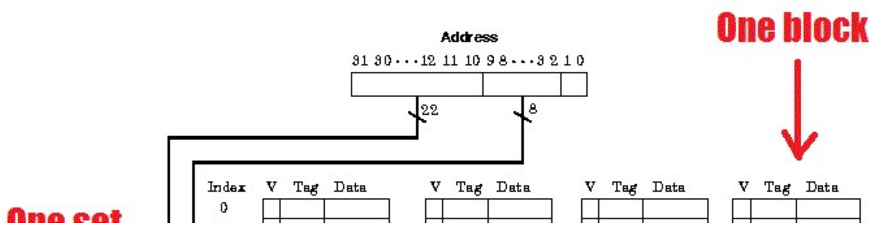
- Cache blocks (or line) contain more than one word
    - For efficiency reasons
    - For spatial locality

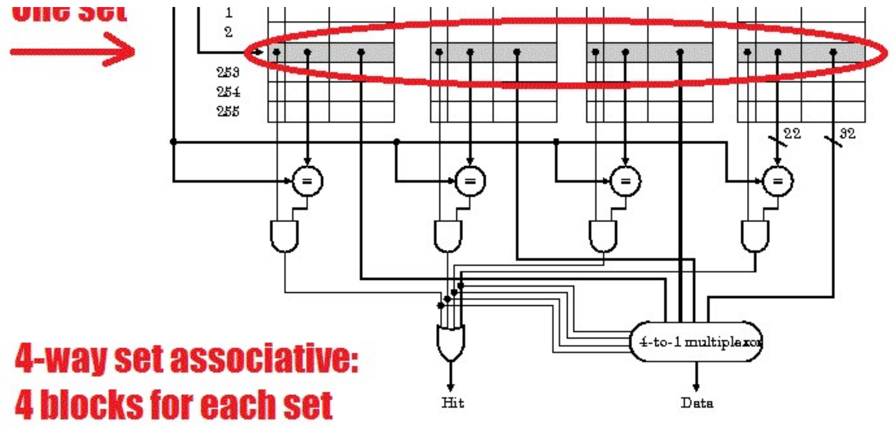
Address division

Tag	Set	Offset

Cache mapping types

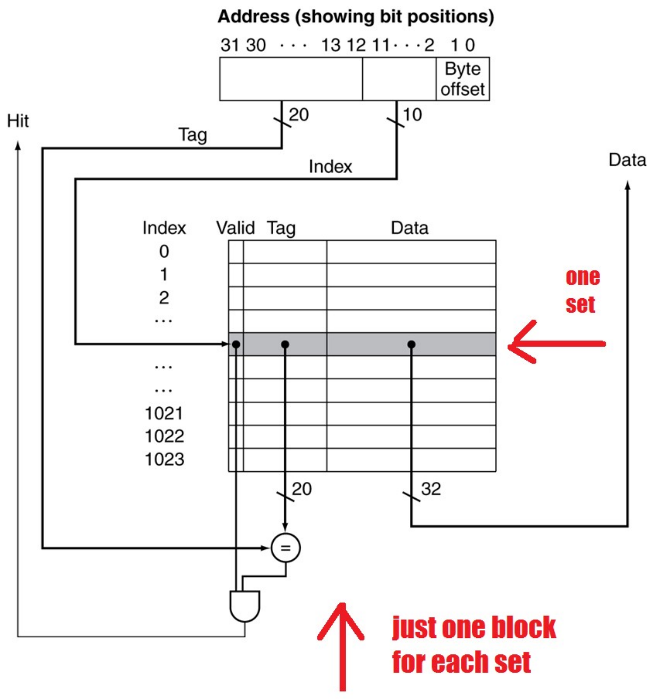
- Set associative
    - Scheme





- Inserting a block consists of
    - Mapping a block onto a set $(Block\ address)\ \%\ \#sets\_in\_cache$
    - The block can be placed **anywhere** within that set
- Finding a block consists of
    - Mapping the block address to the set
    - Searching the set (in parallel) to find the block.

- Direct mapping
    - Scheme



    - $2^{tag\ length}$ sets
    - A block is always placed in the same location
- Fully associative
    - Has just one set
    - A block can be placed anywhere

Avarage memory access time

- $Avarage\ memory\ access\ time = Hit\ time + Miss\ rate \cdot Miss\ penalty$
- Where
    - *Hit time*: time to hit in cache

- *Miss rate:* $\frac{\#misses}{\#accesses}$
- *Miss penalty*: time to replace a block from memory.

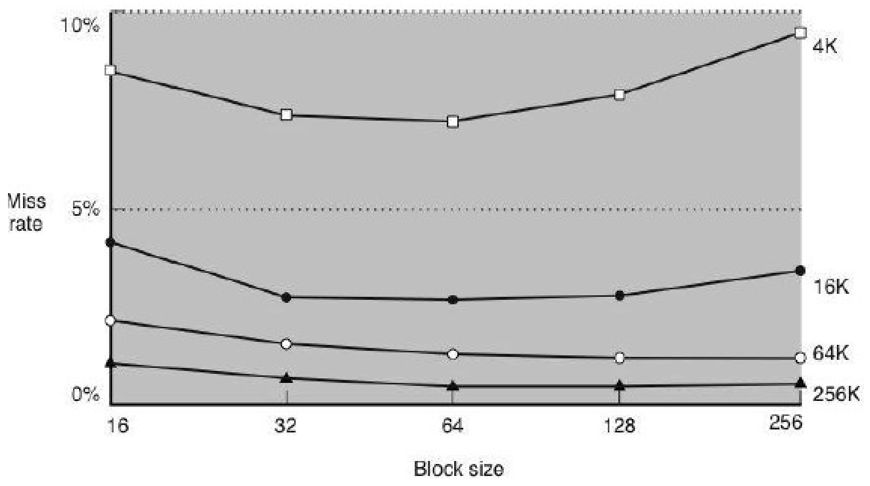
Miss types (3 C's)

- **Compulsory**: the very first access to a block, even with an infinite sized cache
- **Capacity**: reduced to zero w/ a cache large as the main memory
- **Conflict**: during insertion if not fully associative

What happens if

- Larger block size
    - Reduces compulsory misses, hence miss rate (thanks to spatial locality)
    - Increases miss penalty
    - Increases capacity or conflict misses, especially in smaller caches
- Bigger caches
    - Reduces capacity misses, hence miss rate
    - Higher hit time
    - Higher cost and power consumption
- Higher associativity
    - Reduces conflict misses, hence miss rate
    - Higher hit time
    - Higher consumption
- Multi level caches
    - L2 cache leads to
        - Larger blocks
        - Bigger capacity
        - Higher associativity
    - More power efficient
    - $$AMAT = Hit\ time_{L1} + Miss\ rate_{L1} \cdot (Hit\ time_{L2} + Miss\ rate_{L2} \cdot Miss\ penalty_{L2})$$

The ideal block size is the one that considers both aspects:



Why? Recall

Larger block size causes

- Reduces compulsory misses, hence miss rate (thanks to spatial locality)
- Increases capacity or conflict misses, especially in smaller caches

So the effect of capacity or conflict overcomes the compulsory misses effect after a certain block size value.

Instruction and data caches

- Split cache
  - Divided in I-CACHE and D-CACHE
  - More convenient when there's an unbalance between instructions and data cache blocks, because the I-CACHE will produce way less misses.
- Unified cache
  - Everything in one cache type
  - More convenient where there's a balance between instructions and data cache blocks.

Write policy

Both of them can use a *write buffer*, so the cache can replace blocks without waiting for data to be written in main memory

- Write-through
- Write-back

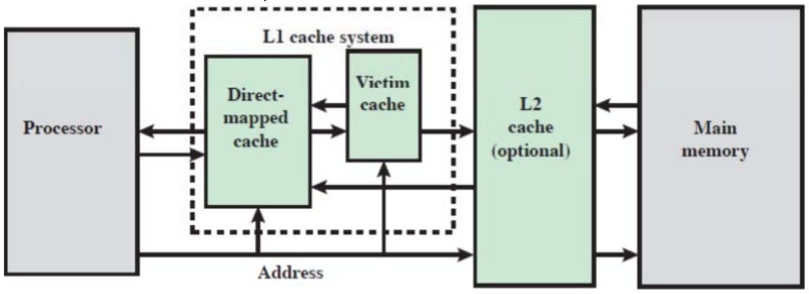
Replacement policy

- Direct mapped cache
  - No choice due to fixed cache blocks position in cache
- Set associative cache
  - Randomly
  - FIFO
  - **LRU** (most used)
    - The cache stores the information about the order of accessed data blocks.
      This info is uptated only after write operations.
  - **Pseudo LRU**
    - Uses a smaller number of bits to mantain the order of accesses data blocks, like in the LRU policy.
  - Round-robin

Cache optimizations

- Way prediction to **reduce hit time**
  - Combines fast hit time of Direct Mapped cache and have the lower conflict misses of 2-way cache.
  - Added to each block of a cache are block predictor bits.
    The bits select which of the blocks to try on the next cache access.
  - If the predictor is correct, the cache access latency is the fast hit time. If not, it tries the other block, changes the way predictor, and has a latency of one extra clock cycle.
  - This prediction means the multiplexer is set early to select the desired block, and only a single tag comparison is performed that clock cycle in parallel with reading the cache data.
- Pipeline cache access to **increase cache bandwidth:**
  - One memory value can be accessed in Cache at the same time that another memory value is accessed in DRAM.
- Nonblocking cache
  - Allows cache hits before previous misses complete
  - Only L1 miss penalty can be hidden
  - Requires out-of-order execution
- Multibanked caches
  - Cache as indipendent banks to support simultaneous access
- Compiler optimizations
  It changes the software organization in order to reduce access time
  - ☐ Loop interchange (nested loops)

        - Puts different accessed variables into a fewer amount of cache blocks in order to reduce misses as much as possible (at least one compulsory miss for each cache block).
        - This is done via software and it basically hasn't any cost.
    - ☐ Blocking
        - Improves temporal locality
- Victim cache
    - Efficient for trashing problem in direct mapped caches
    - L1 and Victim caches are exclusive
    - Miss in L1 and hit in VC; miss in L1 and VC



Multi level cache

- Usually:
    - L1 is set associative for private variables
    - L2 is direct mapped for shared variables
- Two types
    - Inclusive caches
        - L2 contains removed cache blocks from L1
        - Miss in L1, hit in L2 → block replacement b/w L1 and L2
        - Miss in both caches → cache block fetch in both caches
    - Exlusive caches
        - Miss in one cache and hit in the other one → swap among caches

Cache

mercoledì 08 marzo 2017 10:47

- Locality principle
  - Temporal locality
    If a memory location is accessed, there's an high probability that that same location will be accessed again soon.
  - Spatial locality
    The **pre-fetch unity** uses this principle by saving next instruction in the instruction cache.

Memory hiearchy

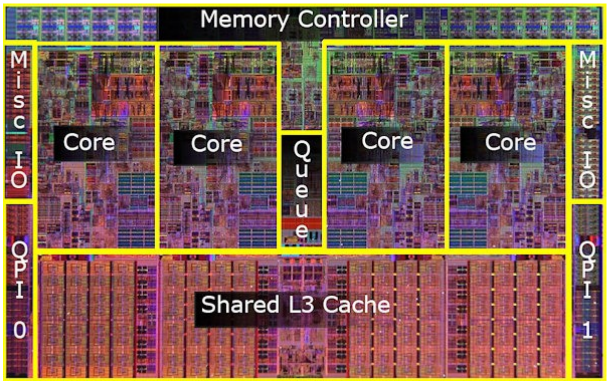
- All cache levels
  - Internal data cache
  - Second level cache (SRAM)
- DRAM (main memory)
- HDD (secondary memory disk)

Using cache: temporal steps

1. The OS (therefore the CPU) loads the software on the RAM
2. The CPU loads the current instruction in cache

Cache levels

Data is searched in the first place in the first cache level: if there's a miss, the research continues in the other caches. When there's a hit, the data block is then copied the the upper cache layer, so, thanks to the temporal locality, if that block will be used again soon, reading it will be much faster.



How CPU loads data in its interal cache

- Direct mapped
  It uses low-order address bits to determine the cache location.
  The operation is: `{ address mod #SetsInCache }`.
  In the cache memory there's also:
  - Tag: that indicates which block is present in a slot.
    Calculation: `floor(address / #BlocksInCache)` (starting from zero obviously)
  - In/valid bit

  An initial state can be like this:

Index (not present in the actual chache)	**Valid bit**	**Tag**	**Data**
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

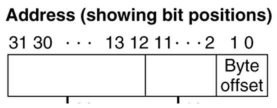
Cache collision

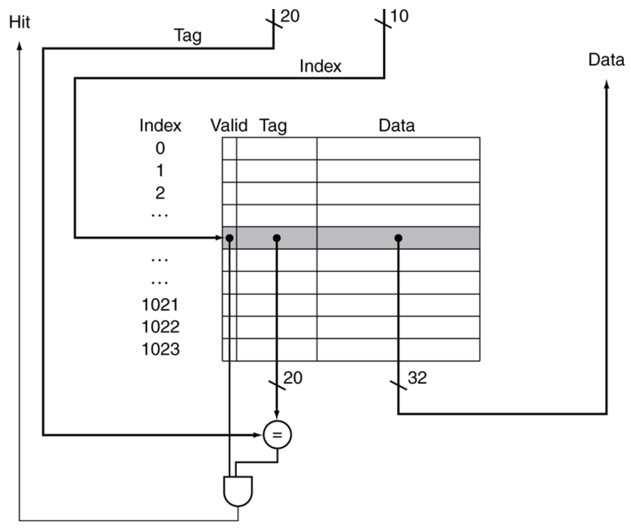
Same low-order address bits block that goes into cache, delets the previous entry.

Address division

tag	**index**	**offset**

Direct mapping schema

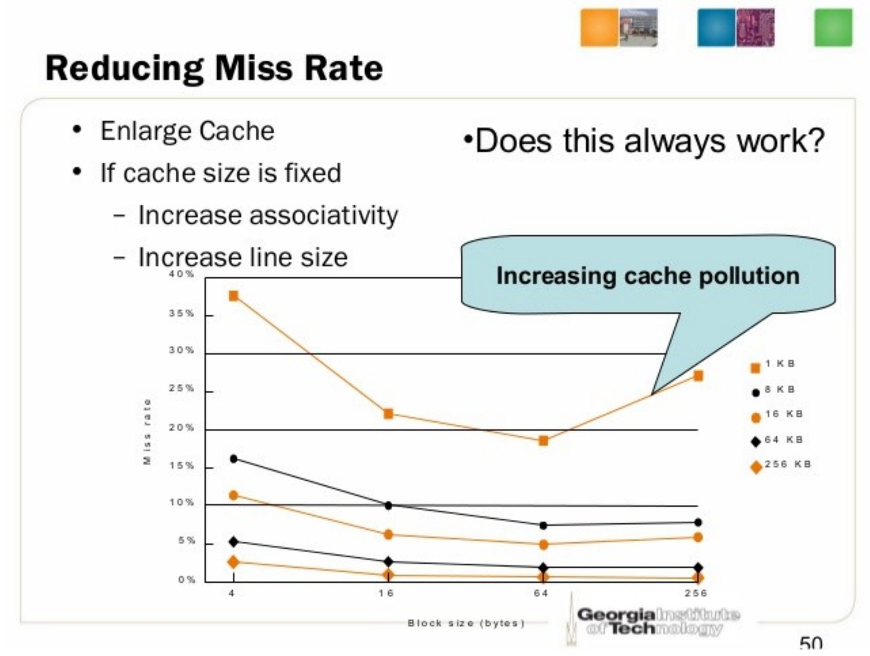




- Byte offset: present becasue there's more than one word in a block.

Blocks size

Blocks size are in contrast w/ the number of blocks in cache.
Goal: reducing the number of misses.
For the temporal locality principle, it's better if blocks are small.



Reducing Miss Rate

- Enlarge Cache
- If cache size is fixed
  - Increase associativity
  - Increase line size

•Does this always work?

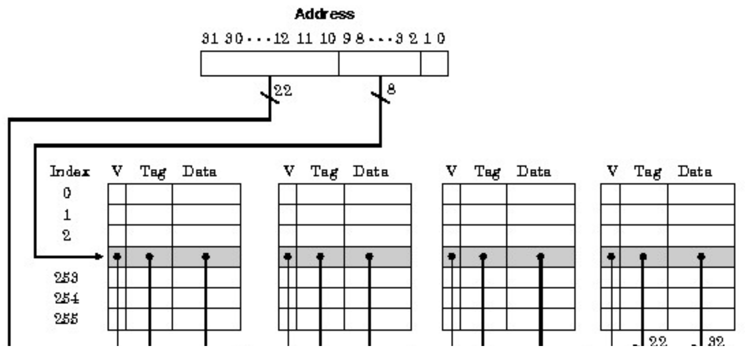
Not always increasing the block size implies a growing number of misses, because of the spatial locality principle.
This type of graphs are produced with benchmarks.
The best value for a block size is the one that minimizes … ? It may depend on the type of the application.

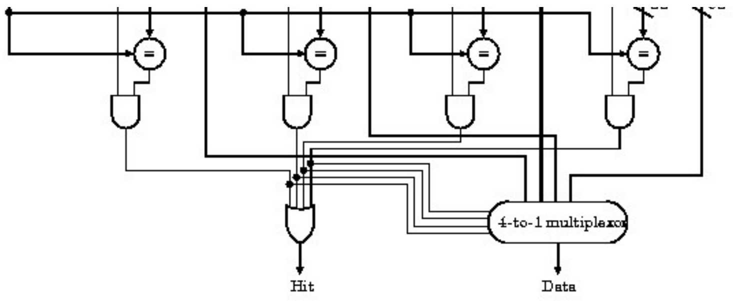
Compulsory misses

Cache misses caused by an empty cache or by accessing a data block in the main memory for the first time.
They can be avoided by pre-loading the cache with …

Associative cache

In a direct mapped cache, a data block can only be present in one cache block.
Other types of caches allows multiple possible locations for a data block, and that decreases the miss rate.
There's an example of an associative cache:

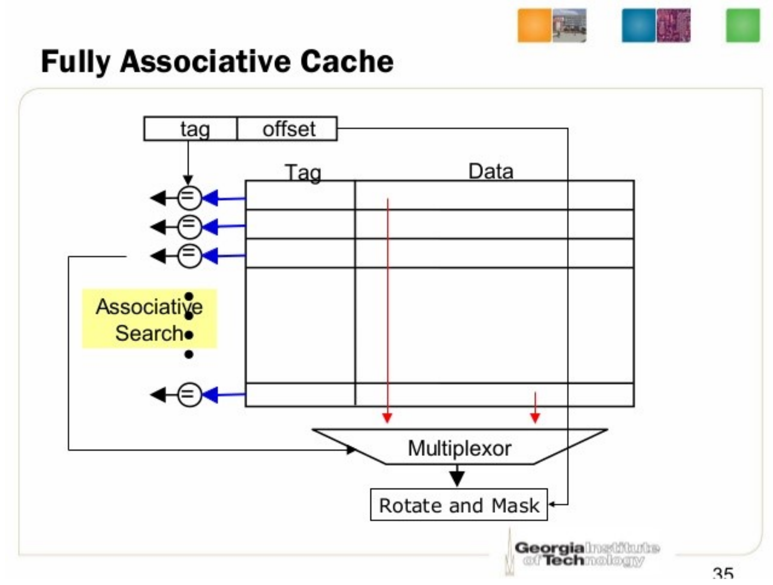




Collisions happen only when all four blocks are full.

Fully associative cache

A data block can be in any cache block.



Conflict miss vs capacity miss

The title is self-explainatory.
It's important to understand the type of miss because it can the useful for the
In order to avoid capicity miss, blocks size has to increase.

Translation Lookaside Buffer (TLB)

It's just a cache used for address translation.
TLBs are fully associative caches.

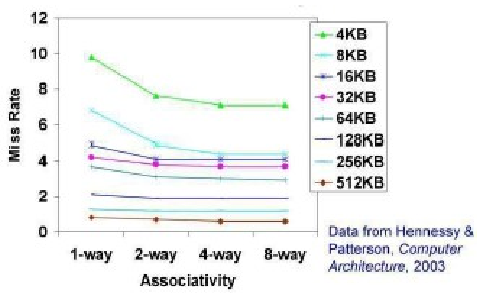
I/O transfers

In those cases the cache is useless 'cause the I/O sensors convert the data and then save it on the HDD.

3 types of misses

- Compulsory miss
  Cannot be reduced to zero, it's compulsory.
- Capacity miss
  Reduced to zero w/ a cache large as the main memory.
- Conflict miss
  Reduced to zero w/ a fully associative cache.

As shown below, the number of misses can be reduced by increasing the associativity.



Replacement policy

- Direct mapped cache
  There's not choice, 'cause the data block in the main memory has a pre-fixed position in the cache.
- Set associative cache
  - Randomly

Gabriele Ara's notes

-compulsory miss cannot be avoided (costant fo
- capacity miss caused by the used of small size
- conflict miss when two blocks have to be place

moving to associative cache reduces conflict mis
increasing the size of the case I reduce the capa
we can reduce the miss byut we have to pay in s
high)

when we have a miss in cache the cpu access m
cache with hte new block.
if the block was occupied we have to replace the

- for direct mapped we have no choice, the bloc
when we use a set associative cache we can use
- the first choice of policy is random but obvious
- the most used policy is least recently used bec
accessed
the information about the "use" is changed duri
the solution to update this information should b

typically it is used a pseudo LRU policy, in order

until now we only talked about read operations,

If I have write operation I have to update the va
there are 3 policies in priciple:

- FIFO
- **LRU** (most used)
  The cache stores the information about the order of accessed data blocks.
  This info is uptated only after write operations.
- **Pseudo LRU**
  Uses a smaller number of bits to mantain the order of accesses data blocks, like in the LRU policy.
- Round-robin
- ...

Coherency b/w cache and main memory

This problems comes up dring a data-write hit.
Possible choices:

- Write through
  The main memory is immediately updated.
  Updating the main memory though is a slow operation.
  To overcome this problem:
  - Write buffer
    An SRAM buffer permits the CPU to continue with its operations.
    There's a stall only when the write buffer is full.
- Write back
  Just update the cache block.
  When the cache block has to be replaced, the cache controller has to update the data block in the main memory.
  Cons:
  - In a multi core architecture, a core updates only its private cache.
    There's a controller (SCU, maybe?) that listens all the operations on the CPU bus, and it can invalidate or update the other core caches.

Write miss

- Write through
  - Allocate and miss
  - Write around

Considering a value used in a for loop, instead of writing in this var n times, the CPU can also update this var by writing its final value: this strategy increases the bus throughput.
The avarage percentage of writing operations in a program is around 20-50%.

Performance evaluation

- AMAT (Avarage Memory Access Time)
  - Hit time: time required by the CPU to know if the data block is present or not in cache.
  - Miss rate
  - Miss penalty: time spent by the CPY in order to acquire the data block from the main memory and put it in cache.

Multilevel cache

Different level caches can be implemented in different types: direct mapped, associative, ...
E.g., private thread vars can bu put in L1 caches, meanwhile shared vars can be stored in a lower-level cache: this is useful in multi cores CPUs.

Different types of multilevel cache

- Inclusive cache
  L1 is included in L2 (it's just another small copy).
  So, the L2 cache contains all the cache blocks removed from L1.
  Of course cache block updates have to be propagated in all levels.
  - When there's a miss in L1 and a hit in L2, the CPU can replace a non-used block in L1 w/ the needed data block present in L2.
  - When there's a miss in both L1 and L2 caches, the CPU has to load the needed data block in both caches.

  The access time for this type of multilevel cache can be calculated w/ the recursive formula shown in slide 13.
- Non-inclusive cache
  Data blocks in L1 are not contained in L2 and vice versa (L2 is a "victim cache" for L1).
  L2 so it's useful to avoid future misses.
  The total size of the cache is the sum of those two.
  - When there's a miss in L1 and a hit in L2, the CPU swaps the needed data block in L2 w/ a unused block in L1.
  - When there's a miss in both L1 and L2 caches, the data block taken from the main memory will be put only in the L1 cache.

Advanced CPU

The bus is accessed not only by the CPU, so it has to wait in order to acquire the bus access: the CPU must be able to interrupt operations.

Different type of cache

- Split cache
  Like in the Cortex-A53 processor:
  - Data cache (I-CACHE)
  - Instruction cache (D-CACHE)

  During the fetch phase, the CPU will read data blocks from the I-CACHE, otherwise it will use the D-CACHE.
  If there's an unbalance b/w data accesses and instruction ones, this is the most convenient architecture because in case there are fewer instructions, the I-CACHE would rarely produce misses.
- Unified cache
  It doesn't have this difference.
  If there's a balance b/w data accesses and instruction ones, this is the most convenient architecture.

So there's no "best architecture" in general, it depends on the type of application that it has to run.
Benchmarks always do the job (consisting for example in sorting algorithms).

write through: update is performed in the same
implement a sort of write buffer in order to per

CU has to await only when the buffer is full.

write back: the block in main memory is update
to be read first)

block need to maintain information about clean

notice: other cpu can invalidate a block from th

write allocation: what if we have a miss during a
block. the cache has to fetch the block from mai

with the write through we have another possibi
loading it in cache and then modifying the value

the policies are:

- for write through i can fetch the block (and the
- for write back i have to fecth normally,. the up

loading in cache and avoiding write through pol
single write) when the block has to be replaced.

NOTE: a bad cache can have 5 % as a miss rate (

The percentages of writes for a program is abou
write.

_ Measuring performances

Important to remember:
AMAT = Hit Time + Miss Rate * Miss Penalty (co

_ Multilevel Cache

We can implement one cache nearest to the CP
outside with a direct mapped architecture)
Another reason can be that we can put for exam
CPU)
So that changes in a thread do not invalid values

Inclusive cache if L1 is a subset of the second ca
If L1 is not a subset of the second cache L2 we t

This is done often in multithread and multiproce

In Inclusive caching, updates in L1 should be cor

In Inclusive caching we can have a miss in L1 and
In L1 we have to replace a block.

If we have a miss in L2 too we have to take the b
first in the L2 cache and then in the L1 cache.

Now let's use Exclusive caching.
In this case, L1 and L2 contain different element
Size(cache) = Size(L1) + Size(L2)

If we have a miss in L1 but a hit in L2, the block
Moreover if we have a miss in L1 and another m
back in main memory.

Let's talk about AMAT.

taccess = thitl1 + pmissl1 * tpenaltyl1 <- (p is a p

tpenaltyl1 = thitl2 + pmissl2 * tpenaltyl2

thus

taccess = thitl1 + pmissl1 * (thitl2 + pmissl2 * tp

_ Advanced CPU

A CPU can execute other operations while waiti
effect of miss on execution time depends on the
Other instructions can be executed using a pipe

Reducing the hit time

_ Victim Cache

Contains only the values that are removed by ar
If we have a miss but we decide to remove thw
faster than retrieving it from main memory.

_ Split Cache

This is a cache divided in two sub caches, one fo
The processor can know what he wants to retrie
am in a read operand or write result the CPU is

A unified cache has both data and instruction.

Both solutions have pros and cons, it depends o
could be better if we have a balancing between

In order to build the perfect system we perform
For example many algorithms normally execute
Also the compiler optimizations for memory acc

Features of a cache to reduce time to hit in the
- smaller cache
- direct mapped cache
- smaller blocks
- for writes:
    - no write allocate - no hit on cache, just v
    - write allocate - to avoid two cycles pipeli

Reduce miss rate:
- bigger cache
- more flexible placement (associativity)
- larger blocks (limited)
- use a victim cache

2.6 Putting It All Together: Memory Hierarchies in the ARM Cortex-A8 and Intel Core i7

This section reveals the ARM Cortex-A8 (hereafter called the Cortex-A8) and Intel Core i7 (hereafter called i7) memory hierarchies and shows the performance of their components on a set of single threaded benchmarks. We examine the Cortex-A8 first because it has a simpler memory system; we go into more detail for the i7, tracing out a memory reference in detail.

This section presumes that readers are familiar with the organization of a two-level cache hierarchy using virtually indexed caches. The basics of such a memory system are explained in detail in Appendix B, and readers who are uncertain of the organization of such a system are strongly advised to review the Opteron example in Appendix B. Once they understand the organization of the Opteron, the brief explanation of the Cortex-A8 system, which is similar, will be easy to follow.

The ARM Cortex-A8

The Cortex-A8 is a configurable core that supports the ARMv7 instruction set architecture. It is delivered as an IP (Intellectual Property) core. IP cores are the dominant form of technology delivery in the embedded, PMD, and related markets; billions of ARM and MIPS processors have been created from these IP cores. Note that IP cores are different than the cores in the Intel i7 or AMD Athlon multicores. An IP core (which may itself be a multicore) is designed to be incorporated with other logic (hence it is the core of a chip), including application-specific processors (such as an encoder or decoder for video), I/O interfaces, and memory interfaces, and then fabricated to yield a processor optimized for a particular application. For example, the Cortex-A8 IP core is used in the Apple iPad and smartphones by several manufacturers including Motorola and Sam-sung. Although the processor core is almost identical, the resultant chips have many differences.

Title-1

Generally, IP cores come in two flavors. Hard cores are optimized for a particular semiconductor vendor and are black boxes with external (but still on-chip) interfaces. Hard cores typically allow parametrization only of logic outside the core, such as L2 cache sizes, and the IP core cannot be modified. Soft cores are usually delivered in a form that uses a standard library of logic elements. A soft core can be compiled for different semiconductor vendors and can also be modified, although extensive modifications are very difficult due to the complexity of modern-day IP cores. In general, hard cores provide higher performance and smaller die area, while soft cores allow retargeting to other vendors and can be more easily modified.

The Cortex-A8 can issue two instructions per clock at clock rates up to 1GHz. It can support a two-level cache hierarchy with the first level being a pair of caches (for I & D), each 16 KB or 32 KB organized as four-way set associative and using way prediction and random replacement. The goal is to have single-cycle access latency for the caches, allowing the Cortex-A8 to maintain a load-to-use delay of one cycle, simpler instruction fetch, and a lower penalty for fetching the correct instruction when a branch miss causes the wrong instruction to be prefetched.

The optional second-level cache when present is eight-way set associative and can be configured with 128 KB up to 1 MB; it is organized into one to four banks to allow several transfers from memory to occur concurrently. An external bus of 64 to 128 bits handles memory requests. The first-level cache is virtually indexed and physically tagged, and the second-level cache is physically indexed and tagged; both levels use a 64-byte block size. For the D-cache of 32 KB and a page size of 4 KB, each physical page could map to two different cache addresses; such aliases are avoided by hardware detection on a miss as in Section B.3 of Appendix B.

Memory management is handled by a pair of TLBs (I and D), each of which are fully associative with 32 entries and a variable page size (4 KB, 16 KB, 64 KB, 1 MB, and 16 MB); replacement in the TLB is done by a round robin algorithm. TLB misses are handled in hardware, which walks a page table structure in memory. Figure 2.16 shows how the 32-bit virtual address is used to index the TLB and the caches, assuming 32 KB primary caches and a 512 KB secondary cache with 16 KB page size.

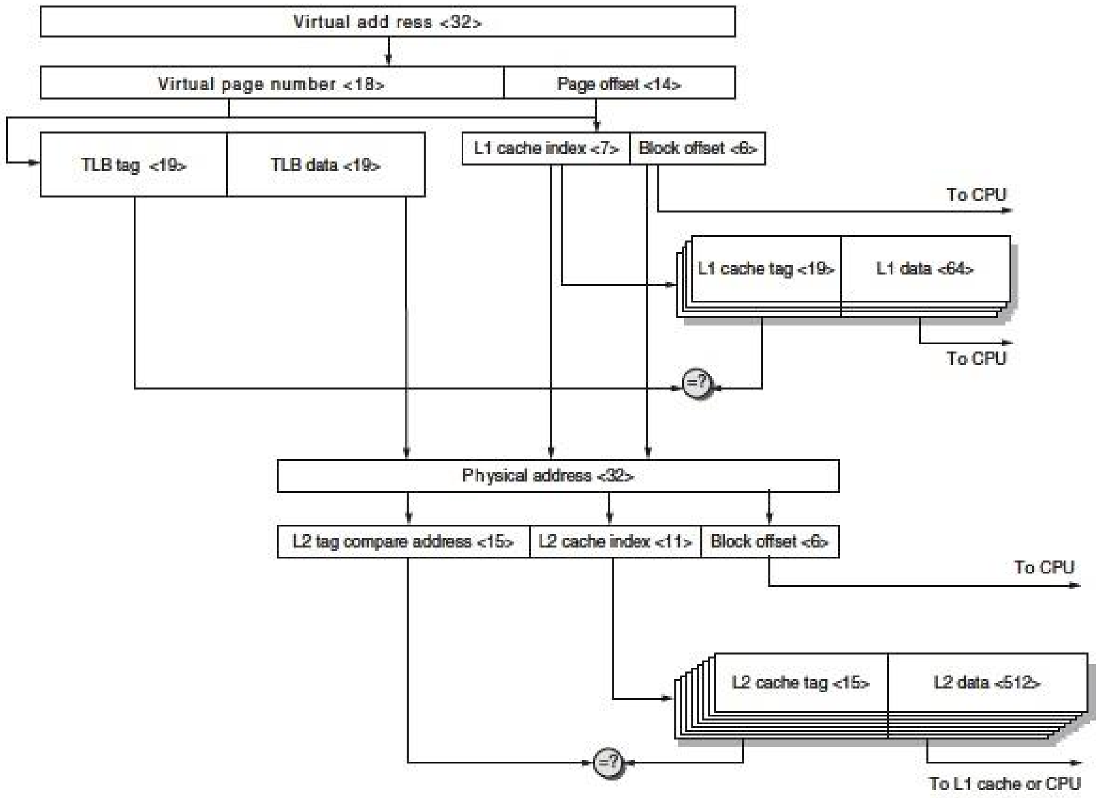


Figure 2.16. The virtual address, physical address, indexes, tags, and data blocks for the ARM Cortex-A8 data caches and data TLB. Since the instruction and data hierarchies are symmetric, we show only one. The TLB (instruction or data) is fully associative with 32 entries. The L1 cache is four-way set associative with 64-byte blocks and 32 KB capacity. The L2 cache is eight-way set associative with 64-byte blocks and 1

MB capacity. This figure doesn't show the valid bits and protection bits for the caches and TLB, nor the use of the way prediction bits that would dictate the predicted bank of the L1 cache.

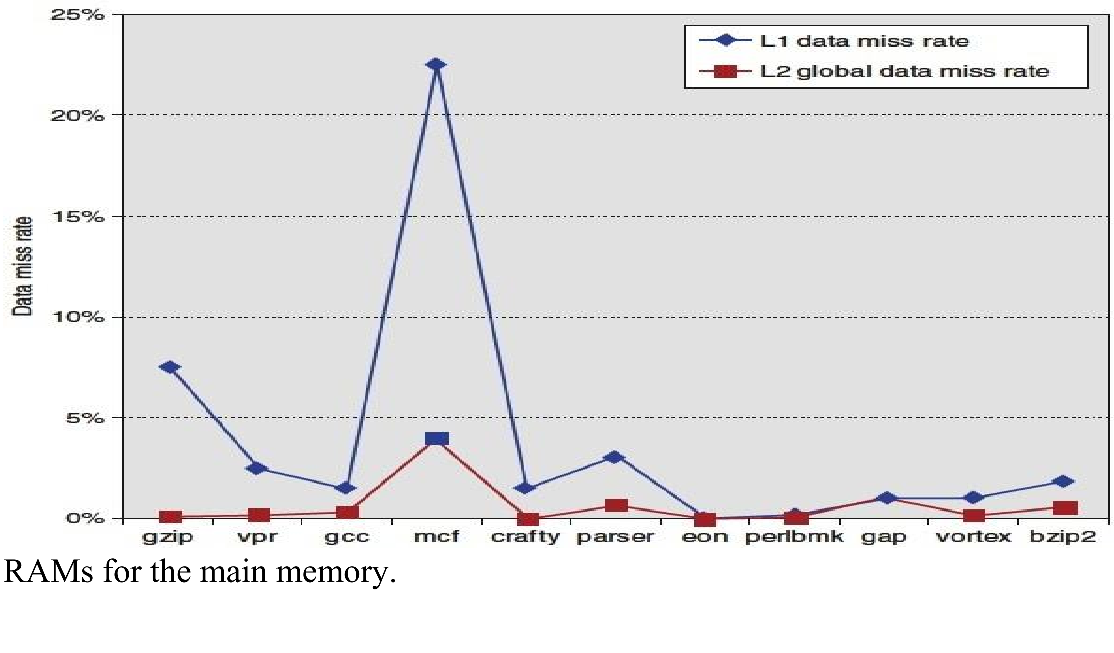
Performance of the Cortex-A8 Memory Hierarchy

The memory hierarchy of the Cortex-A8 was simulated with 32 KB primary caches and a 1 MB eight-way set associative L2 cache using the integer Minnespec benchmarks (see KleinOsowski and Lilja [2002]). Minnespec is a set of benchmarks consisting of the SPEC2000 benchmarks but with different inputs that reduce the running times by several orders of magnitude. Although the use of smaller inputs does not change the instruction mix, it does affect the cache behavior. For example, on mcf, the most memory-intensive SPEC2000 integer benchmark, Minnespec has a miss rate for a 32 KB cache that is only 65% of the miss rate for the full SPEC version. For a 1 MB cache the difference is a factor of 6! On many other benchmarks the ratios are similar to those on mcf, but the absolute miss rates are much smaller. For this reason, one cannot compare the Minniespec benchmarks against the SPEC2000 benchmarks. Instead, the data are useful for looking at the relative impact of L1 and L2 misses and on overall CPI, as we do in the next chapter.

Title-3

The instruction cache miss rates for these benchmarks (and also for the full SPEC2000 versions on which Minniespec is based) are very small even for just the L1: close to zero for most and under 1% for all of them. This low rate probably results from the computationally intensive nature of the SPEC programs and the four-way set associative cache that eliminates most conflict misses.

Figure 2.17 shows the data cache results, which have significant L1 and L2 miss rates. The L1 miss penalty for a 1 GHz Cortex-A8 is 11 clock cycles, while the L2 miss penalty is 60 clock cycles, using DDR SD



RAMs for the main memory.

Figure 2.17. The data miss rate for ARM with a 32 KB L1 and the global data miss rate for a 1 MB L2 using the integer Minnespec benchmarks are significantly affected by the applications. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate, that is counting all references, including those that hit in L1. Mcf is known as a cache buster.

Using these miss penalties, Figure 2.18 shows the average penalty per data access. In the next chapter, we will examine the impact of the cache misses on overall CPI.

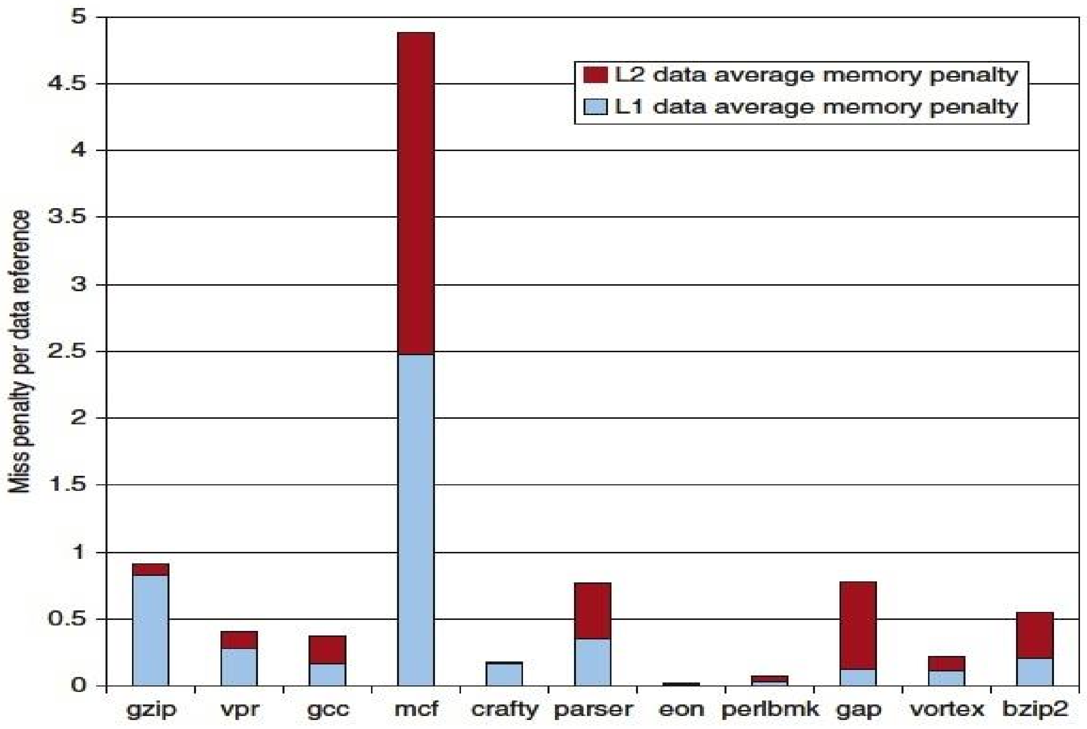


Figure 2.18. The average memory access penalty per data memory reference coming from L1 and L2 is shown for the ARM processor when running Minniespec. Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

Memory hierarchy

venerdì 10 marzo 2017 08:43

Part III

Why multi-level cache

- To save power consumption per bit
- To reduce latency

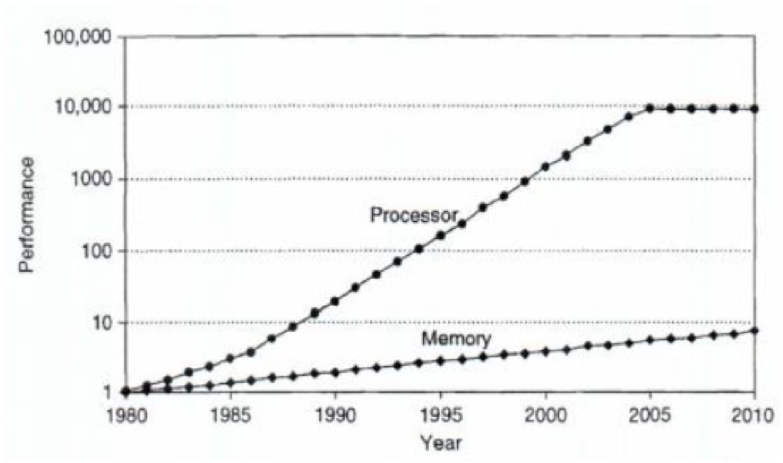
Registers vs caches

- Registers are managed by the program
- Cache is not, it's automatic

On embedded systems

On some MCUs, high speed memories can be used as caches or high speed memory: cache is automatically controlled by the controller, meanwhile the memory is used to contain data and instructions used by the program.
The MCU decides by itself whether to use the memory chip as a cache or as an high speed memory.

Problem: memory much slower than CPU



$$Memory\ acccess\ time = 100 * clock\ cycle\ time$$

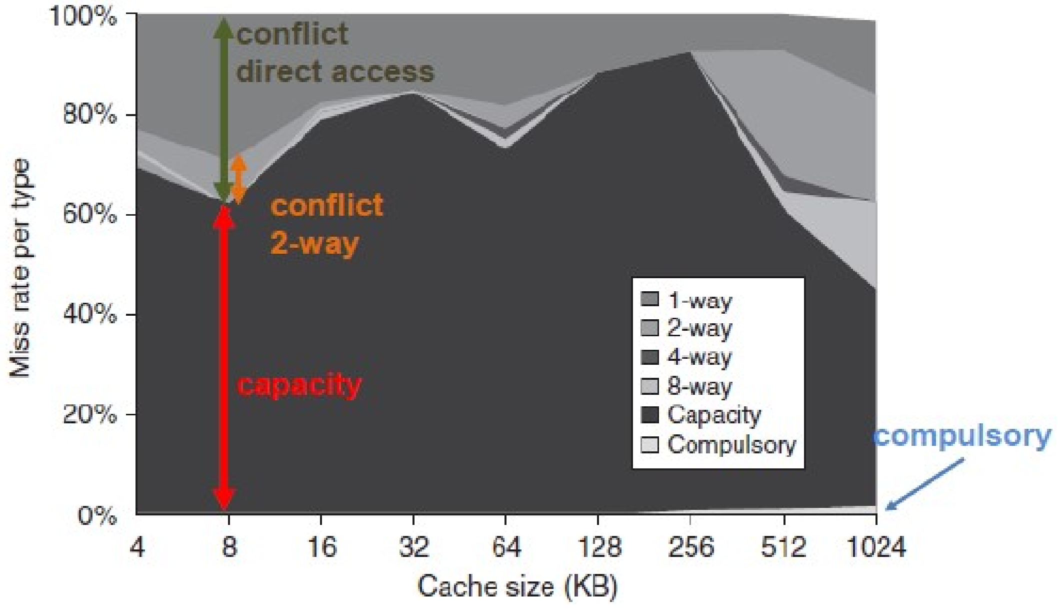
Solution

1. Use of cache in pipeline mode
   - In this mode, one memory value can be accessed in Cache at the same time that another memory value is accessed in DRAM.
2. Multiple levels of cache
3. I-CACHE and D-CACHE
   In the same time, the CPU is able to access both caches.
4. Every core has its own cache (L1), and then there's a shared caches b/w cores (L2)
   It minimizes the conflict in the L1 cache.

Write buffer

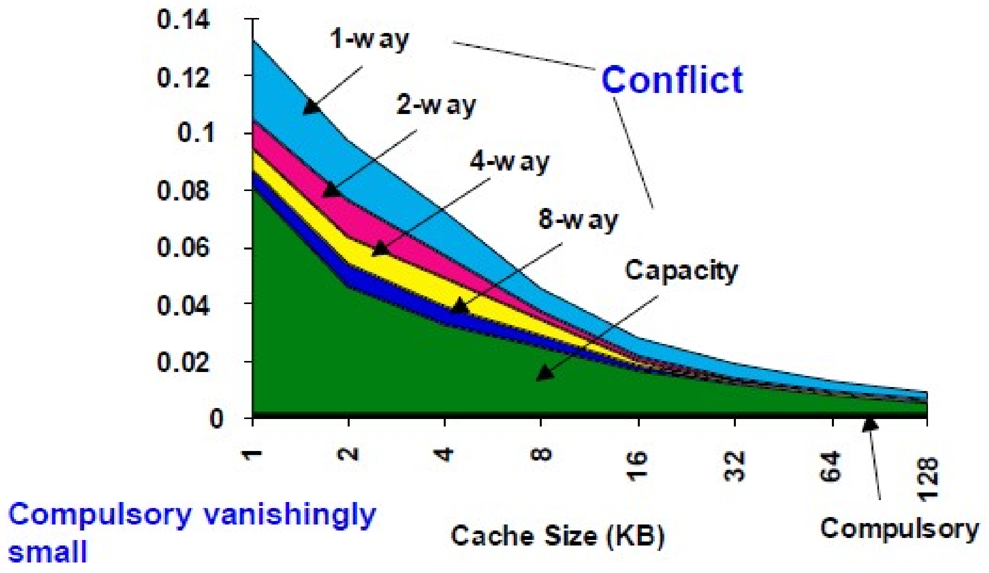
In a multiprocessor environment, the write buffer is called the "invalidation buffer": there's only the address of the word involved in the write operations.

What misses should be minimized?



- The number of conflict misses depends on the cache structure: direct mapped or n-way associative cache. This number of misses depends on the type of the program, so, in an embedded environment, it's a good idea to adapt the cache architecture depending on the program type.
- The capacity miss is the main problem, as shown in the graph above.
  It can be reduced by increasing the cache size, so there's no so much to do.

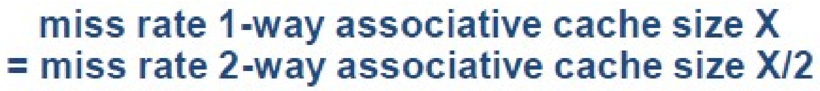
In absolute terms:



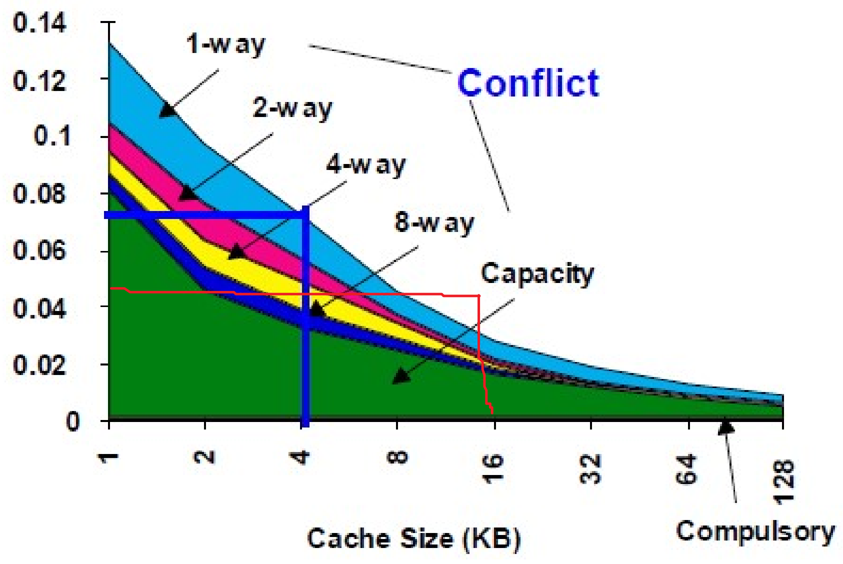
It seems like the 8-way associative cache is the best solution.
Given a certain miss rate, there are multiple choises depending on:

- The cache structure
- The cache size

As shown below:



miss rate 1-way associative cache size X
= miss rate 2-way associative cache size X/2



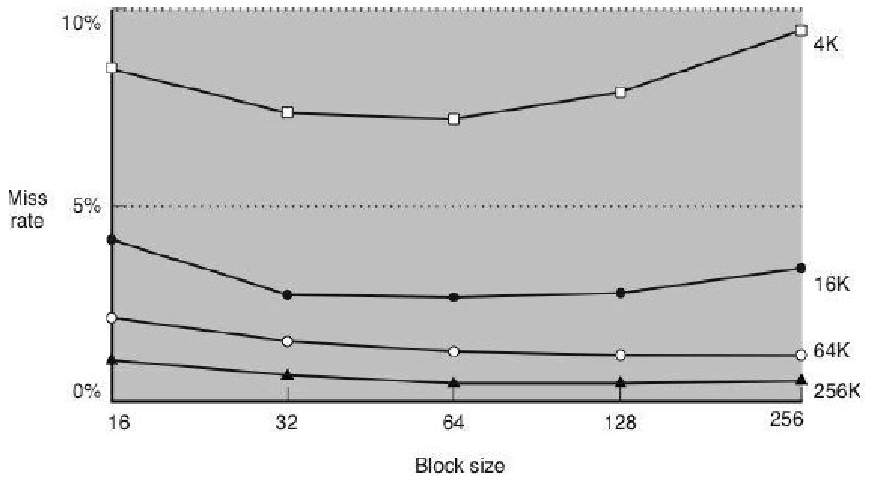
There's also to consider the power consumption: the 8-way associative cache consumes more.

Miss rate vs. block size

With temporal locality, more blocks (w/ small size) are needed.
The increase of block size will however support the spatial locality.

The overall performance depends on the global cache size of course.

The ideal block size is the one that considers both aspects:



Why? Recall

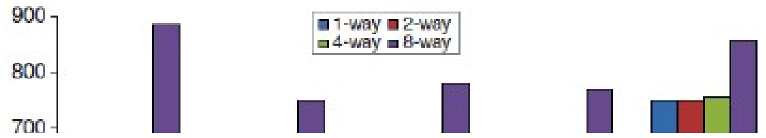
Larger block size causes

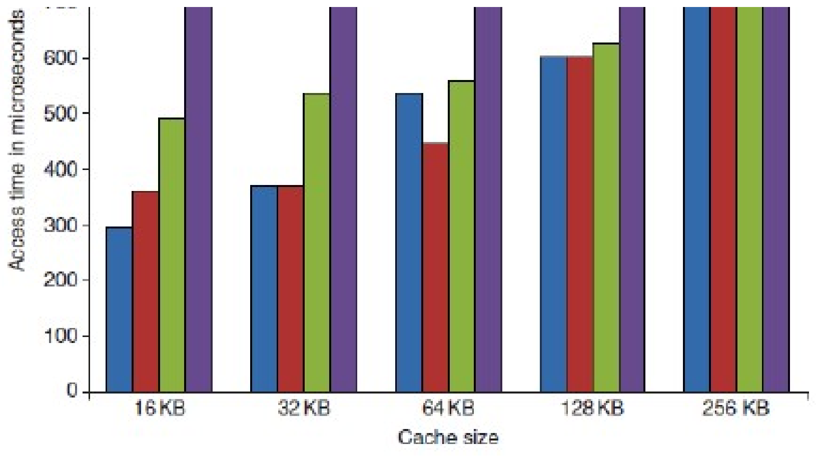
- Reduces compulsory misses, hence miss rate (thanks to spatial locality)
- Increases capacity or conflict misses, especially in smaller caches

So the effect of capacity or conflict overcomes the compulsory misses effect after a certain block size value.

Also here it's important to consider:

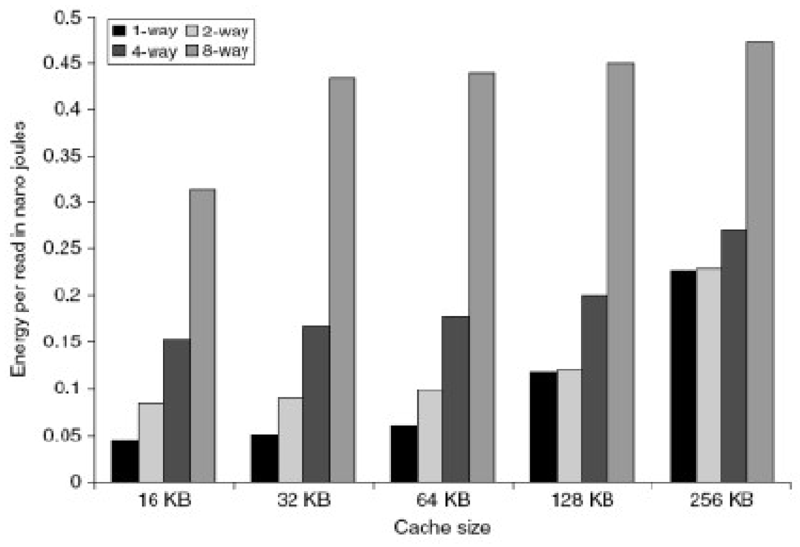
- Power consumption
- Access time





The above graph shows that the 8-way associative cache is so much slower than the 1-way cache.
In big caches (right part of the graph) this difference doesn't really matter.
This graph also **depends on the type of the application**.

Power consumption is directly proportional to the access time:



The direct access cache is the best idea in this case.
Also this graph depends on the type of the program.

It's not easy though to choose the best cache architecture: it depends on a lot of factors, and the best way do determine it is to run a bunch of related benchmarks.

? Cache predictor

The compiler doesn't know the cache organization in compile-time.

? It's important to offer this knowledge on embedded applications because

☐ See wikipedia and the book

? Multibanked caches

Compiler optimizations

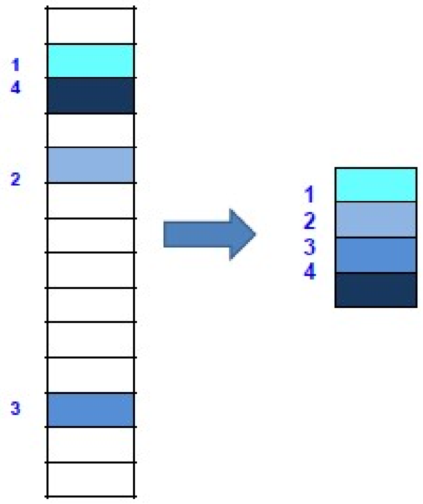
It changes the sw organization in order to reduce access time.

- Loop interchange
  It's possible to copy the array in another one using a better element order to minimize the number of cache misses.

1. ?
   - ☐ Check out the PDF for an example (and copy it)

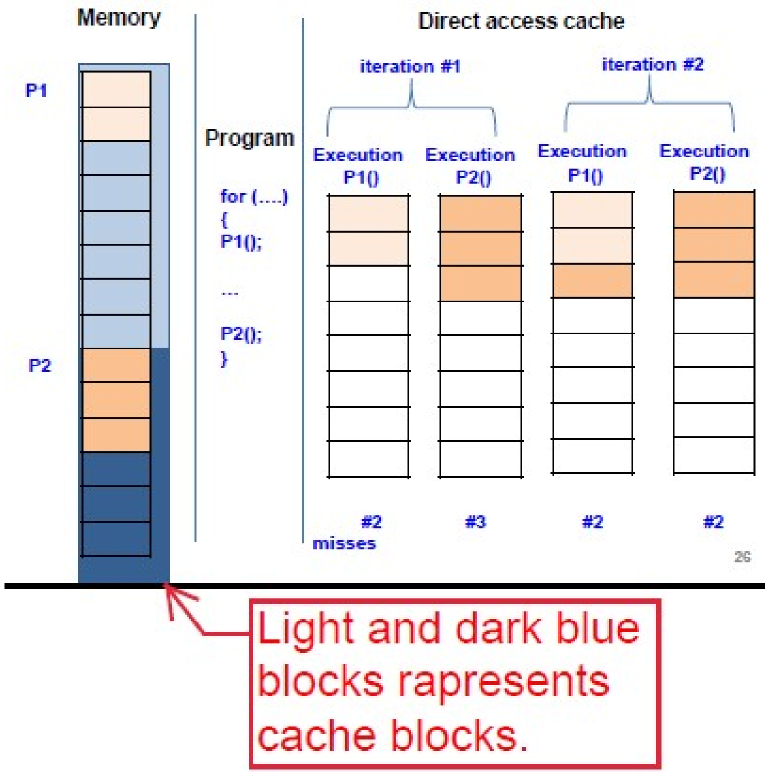
2. Improving temporal locality

If the compiler is able to know the typical sequence of accessed variables, it can move these in order to fit them all in a smaller number of cache blocks.



3. Conflict misses

Those two programs P1 and P2 are stored in different areas in the main memory.



The program will then require two cache blocks.
P1 is overlapped w/ P2 as shown in the program structure.

? The problem is
? There are 2 different approaches:

4. Software solution
   - ☐ See the PDF.

ARM Cortex-A8

venerdì 10 marzo 2017 10:03

Very basic microprocessor for embedded systems.

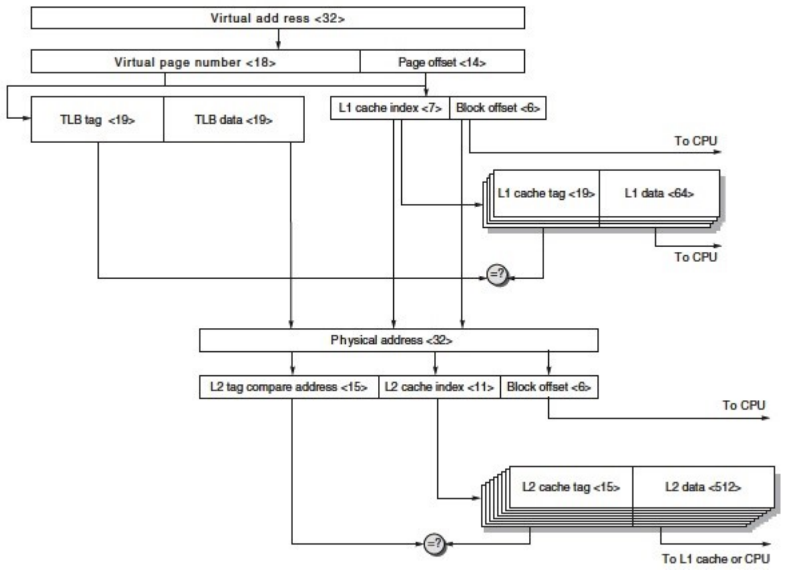
Global organization

- **Translation phase**

☐ Calcolatori elettronici -> virtual address translation

☐ Read http://www.embedded.com/print/4399193

The virtual address is converted.
The virtual page number is used to access a special cache (TLB) that will offer a translation or it can produce a miss.
This is crucial in real-time applications 'cause this operation could require up to 10000 clock cycles.



The physical address is only used when accessing the main memory, so every other cache can use virtual addresses, without the need of translating them.

? The virtual address is defined at compile-time …

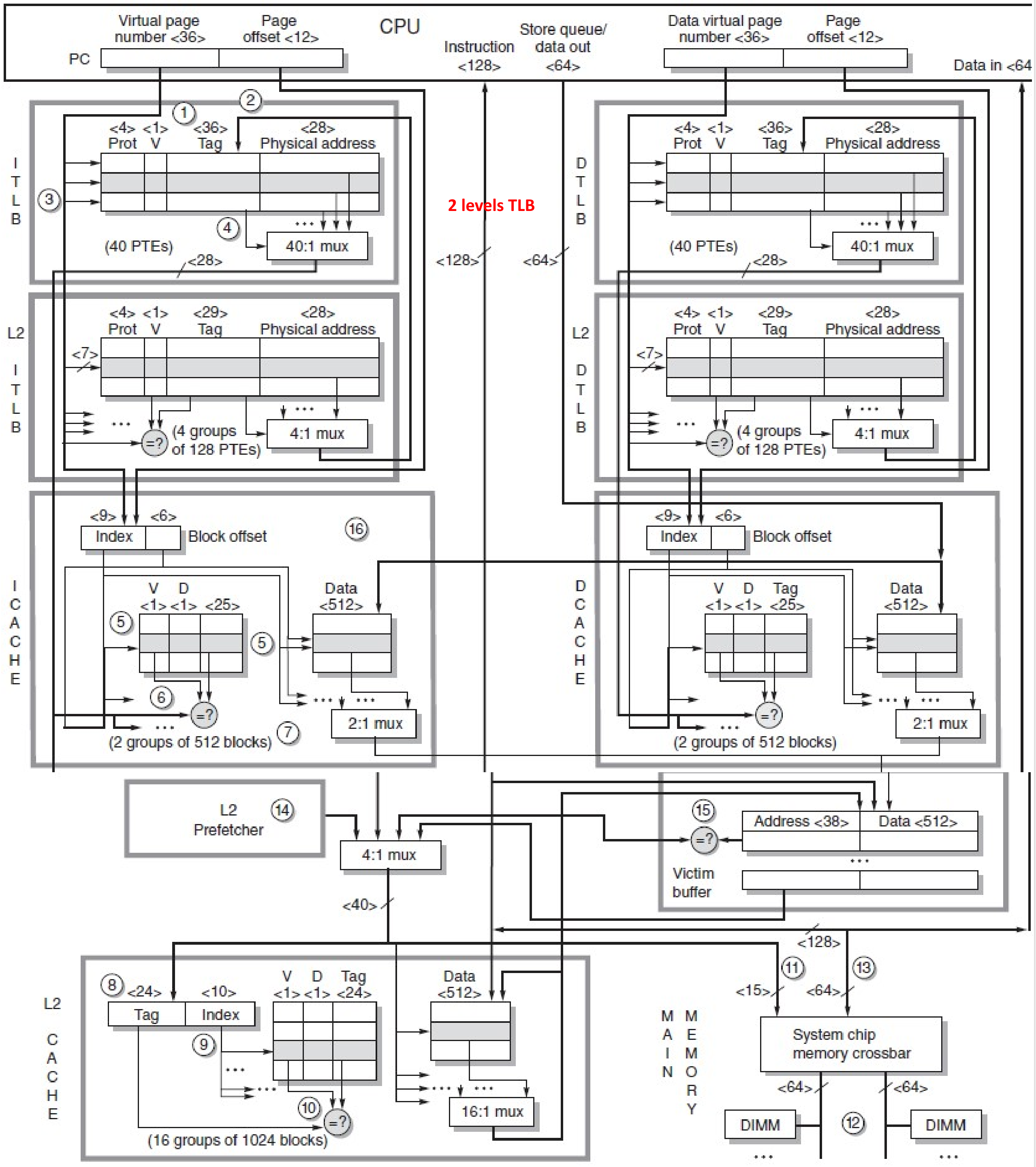
? Different processes can reach shared variables and kernel routines (?) with the same physical addresses by using different virtual addresses.

? There's a way to access the cache and in the same time transalating the address: by checking if the tag in the cache is the same

Intel i7 memory hie rarchy

venerdì 10 marzo 2017 10:29

https://it.wikipedia.org/wiki/Non-Uniform_Memory_Access



GPUs vs CPUs

mercoledì 15 marzo 2017 10:42

What does the GPU do?

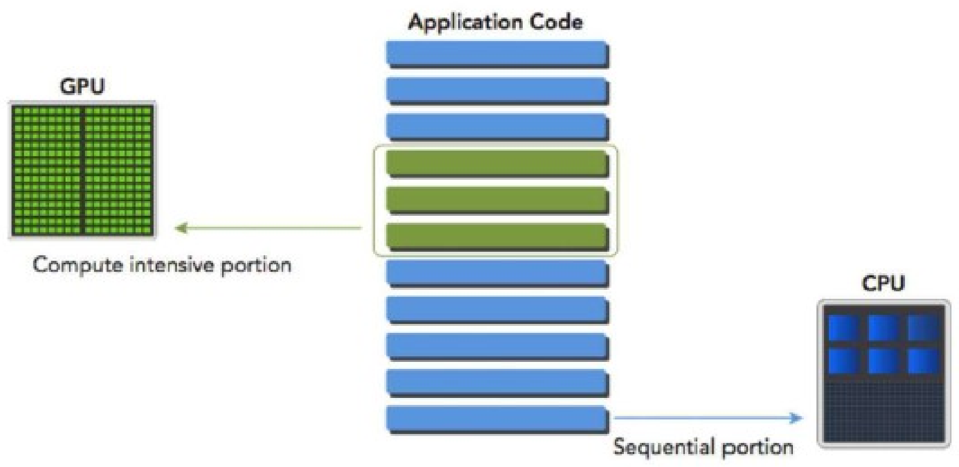
1. Video management
2. Parallel computation with the CPU

GPU cores

A GPU has a thousand of cores: graphics operations are extremely simple, but they are a lot of them (just think of a matrix of pixels, for example…).
E.g., adjusting a pixel brightness is a very simple operation but it has to be run on a large amount of data (pixels).
GPUs does not have to handle so many branches, so it doesn't have the branch predictor.

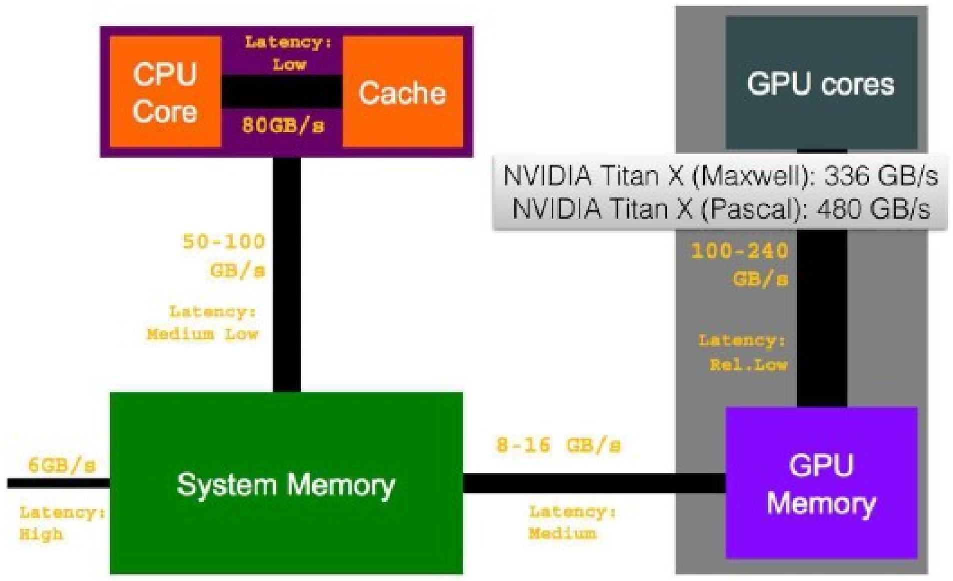
CPU/GPU parallelism

Software can be designed to assign the parallelizable part to the GPU (**device code**) and the sequential part to the CPU (**host code**). This is called an **heterogenous software**.



In the above image, the green part can be performed in parallel and it is run by the GPU (that kinda acts like a co-processor).

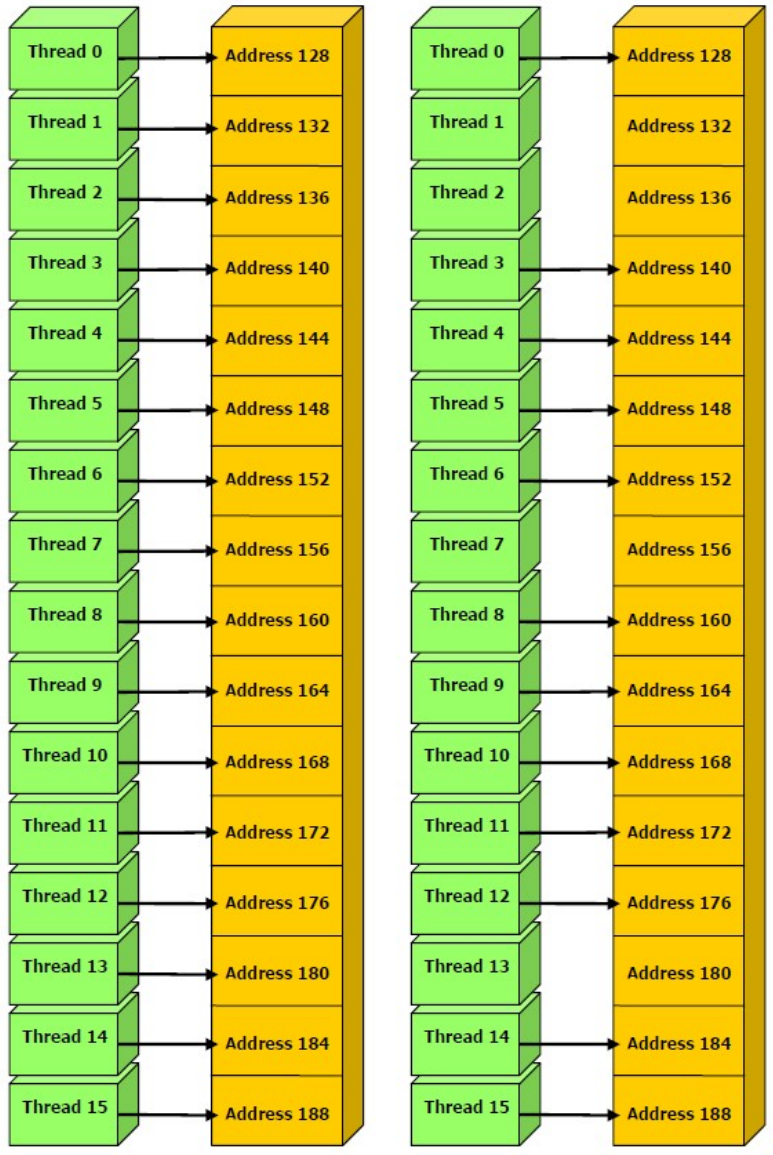
Bandwidth



The latency is the time taken from the CPU to acquire a data block from the cache.

The GPU has its own private memory.
Every GPU core has its own cache; that's why the GPU cache is way more faster than the CPU one: because every core-to-cache bandwidth has to be multiplied by the number of GPU cores.

Coalesced access



Above, there's an example of a coalesced access.
This type of access is perfomed way more faster than a non-coalesced one.

Just increasing the number of cores isn't a solution

The main bottleneck consists in the difference b/w the ...
The goal is to minimize the difference b/w those bandwidths, by:

- Using caches
- Private link among processors
  Like creating
- Minimize the thoughput of processes to the memory subsystem
  By sharing some copies of memory caches among processors.
  This can be done by let every processor run the same instruction in the same type of different data (parallelizable software).
  This is very hard in reality, but in graphic processing it happens quite often.
  What can be done furthermore:

a. Some processors will share the same instruction
b. A set of these groups can then share the same code

Each processor inside a group can shared a set of variables (local cache) and all the processors (cores) can share data in a shared cache (L2).

Solutions

- Software divided in a parallel and a sequential part.
- Split processes into two groups of threads:
  a. One sharing an instruction
  b. The other one, sharing pieces of code

  Target: minimize the communication b/w threads belonging to different processes, and to maximize the communications b/w ones belonging to the same process.

PLX

New executable format that is interpreted as a script (not compiled) by the GPU driver: this solution is useful in order to change the architecture

Why the CPU doesn't have all these cores

The CPU is engineered to run sequential instructions, so CPU cores run at high frequency: GPUs ones run at slower frequencies.
That's because an architecture having a lot of cores consumes a lot, that's why GPUs have all those massive cooling systems.

Threads vs processes

Threads are lighter than processes because the switch context is faster because there's no need to invalidate the TLB, the cache and so on. Changing the virtual table register is not a big deal.
Threads though do not have a lot of synchronization *skills*.

The GPU is a SIMD device

The CPU is more of a SISD engine. Meanwhile, the GPU is a SIMD engine because it has to perform a single instruction to muliple data at the time.

SISD vs SIMD

SISD	**SIMD**
• n clock cycle to fetch and decode the instruction • n clock cycle to acquire the data from the cache memory • n clock to perform the execution of the instruction • n clock cycle to write back in memory (or in the register) the result	• 1 clock cycle to fetch and decode the instruction • n clock cycle to acquire the data from the cache memory • 1 clock to perform the execution of the instruction • n clock cycle to write back in memory (or in the register) the result

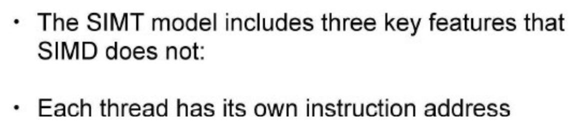
There are currently around 400 SIMD instruction, so it's a very limited set.
This set of instruction is produces by the compiler, so it's not under the developer's control.

SIMT

NVIDIA calls this architecture *Single Instruction Multiple Threads*.

SIMT vs SIMD

In a SIMT architecture there could not be any branch conditions into loops.
In a SIMD architecture, instead, this is possible.



- The SIMT model includes three key features that SIMD does not:

- Each thread has its own instruction address counter.

- Each thread has its own register state.
- Each thread can have an independent execution path.

CUDA: Compute Unified Device Architecture

- It enables a general purpose programming model on NVIDIA GPUs. Current CUDA SDK is 8.0.
- Enables explicit GPU memory management
- The GPU is viewed as a compute **device** that:
  - Is a co-processor to the CPU (or **host**)
  - Has its own DRAM (global memory in CUDA parlance)

    Runs many threads in parallel
- NVIDIA's CUDA nvcc compiler separates the device code from the host code during the compilation process.

- [ ] Install NSight IDE

- A typical CUDA program structure consists of ve main steps:

Operations 2 and 4 are not fast.

1. Allocate GPU memories.
2. Copy data from CPU memory to GPU memory.
3. Invoke the CUDA functions (called by NVIDIA **kernel**) to perform program-specific computation.
4. Copy data back from GPU memory to CPU memory.
5. Destroy GPU memories.

The key is to overlap the initial piece of code with the data copy operation (number 2) from the CPU to the GPU.
A typical application that can use this advantage is a continous processing program on data flows.

CPU/GPU sync

The CPU has to know when the GPU has finished running the *device (parallel) code*.

Grids

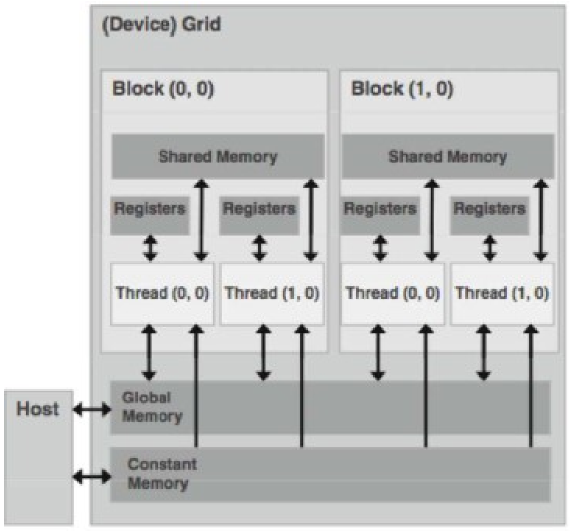
- The programmer decides how to organize a **grid**, to improve parallelization
- When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is invoked.
- Grids are organized into **blocks**.
  - Grids can be 2D or 3D

- Blocks can be identified by coordinates

Hello world example

- [ ] See the slides

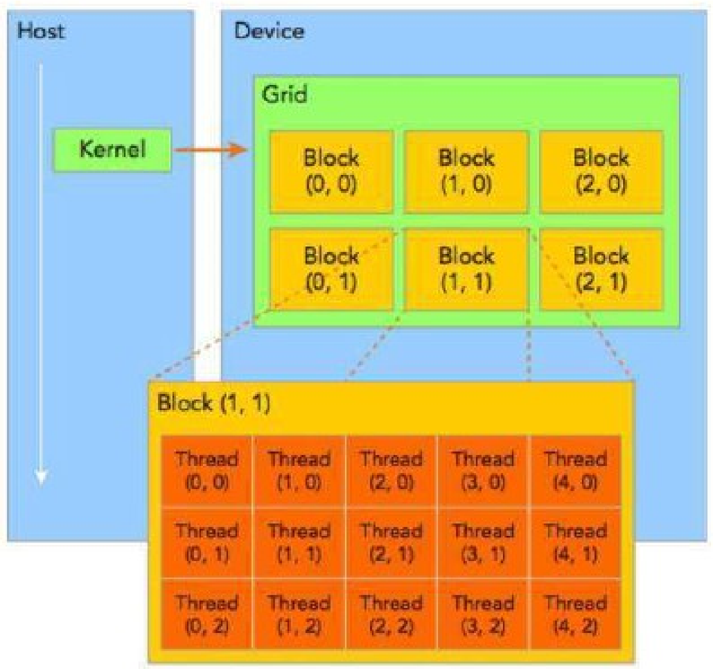
CUDA device memory model



- **Global memory**: used by cudaMalloc()

Matrix calculations

It's convenient to organize the threads for a block by organizing them using the same structure used in the matrix, as the following picture shows:



Virtual memory

venerdì 17 marzo 2017 14:00

Book reference

C4

What is it useful for

- A process' virtual space can be bigger than the actual physical one.
  The RAM memory becomes a sort of HDD cache in this way.
- Processes can share code.
- Accessing kernel objects from user and system processes.
  For a same physical location, different processes can address it with different virtual addresses.
  Since every process has its own virtual address space, every process must have its own page directory.

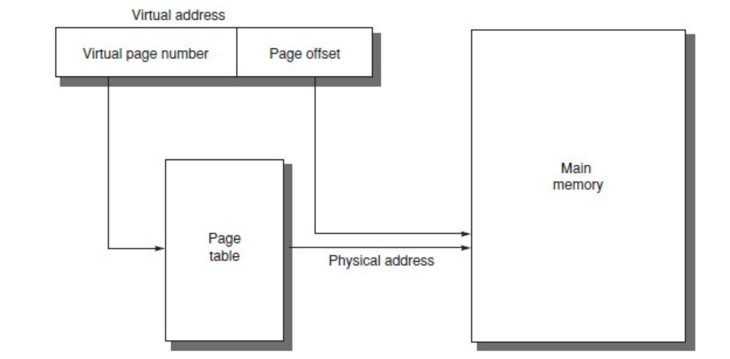


Figure C.22 The mapping of a virtual address to a physical address via a page table.

- Processes can share the same code by having the same translated virtual address: in this way, only one copy is mantained in the main memory.
- The MMU (Memory Management Unit) translates virtual addresses into physical ones: the TLB is just a cache for the page directory.

Translation operation

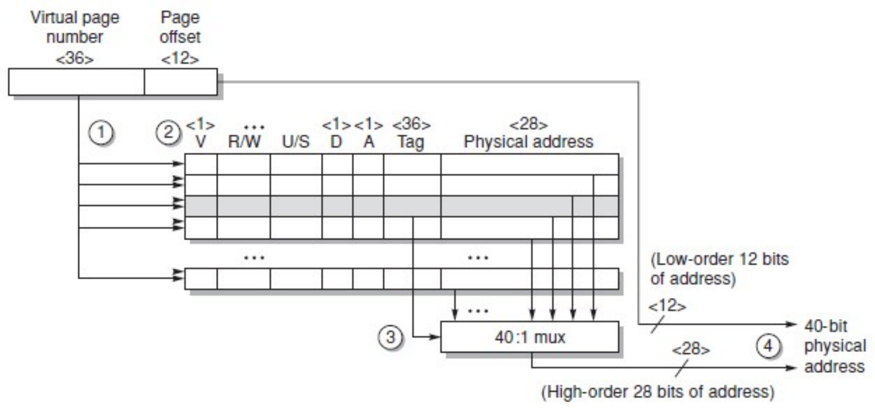


Figure C.23 Operation of the Opteron data TLB during address translation. The four steps of a TLB hit are shown as circled numbers. This TLB has 40 entries. Section C.5 describes the various protection and access fields of an Opteron page table entry.

- R/W flag: read-only or not.
- U/S: in which state the page can be accessed.

Virtual memory advantages

- A protection scheme restricts processes to the blocks belonging only to that process.
- It reduces the time to start the program since not all code and data need to be in physical memory before a program can begin.

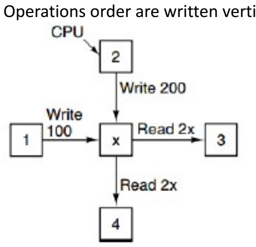
Cache coherence

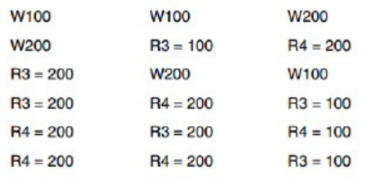
venerdì 28 aprile 2017 08:54

Consistency vs coherence

It's not important to complete the cache blocks updating operations quickly: this can be done as soon as the shared internal CPU bus is free. The important thing is to guarantee the minimum read time.

- **Coherence** [*protocol*]: defines **which values** can be returned by a read operation.
  Any read operation has to return the correct value.
- **Consistency** [model]: defines **when a value** that has been written **will be returned by a read operation**.
  On a read miss, the consistency protocol defines …
  ? a.
  ? b. Memory is not consistent in every moment.
  Write operations are performed in out-of-order
  Different types:
  - **Strict** consistency: any read to a location *x* always returns the value of the most recent write to *x*.
    - ? Hard to implement in a multi-processor (maybe he meant "multi-computer") architecture.
  - **Sequential** consistency: in the presence of multiple read and write requests, some interleaving of all the requests is chosen by the hardware (nondeterministically), but **all the CPUs have to see the exact same operations order**.
    - Example
      Operations order are written vertically





W100	W100	W200
W200	R3 = 100	R4 = 200
R3 = 200	W200	W100
R3 = 200	R4 = 200	R3 = 100
R4 = 200	R3 = 200	R4 = 100
R4 = 200	R4 = 200	R3 = 100

  - **Processor** consistency
    Different CPUs can see different operations order.
    i. Writes by any CPU are seen by all CPUs in the order they were issued.
    ii. For every memory word, all CPUs see all writes to it in the same order.
  - **Weak** consistency: no order in write operation but synchronization point.
  - **Release** consistency: write before a release

Informally, we talk about coherence memory system if any reading of a data provides the value of the written data more recently.

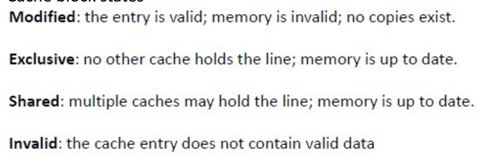
- Write-back cache
  The cache block can be written only when it is needed and when the shared internal CPU bus is free.
- Write-through cache

- Write-update protocol
- Write-invalidate protocol

On 100 memory accesses, on avarage there can happen like 20 writes.
Of those, there could be an even lower number of misses, like 3 o 4.

Coherency protocols

- Used to solve the problem of **consistency**.
- Two approaches
  - **Snooping**-based protocols
    All the operations on the shared bus are observed by cache controllers (snoopers).
    - This solution though is **not scalable** (if the shared L2 cache is not inside the SoC, even 4 processors is a problem).
    - Protocols
      - **MESI**
        - Cache block states



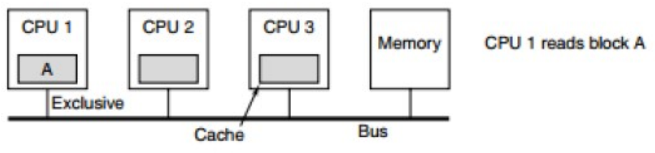
Modified: the entry is valid; memory is invalid; no copies exist.

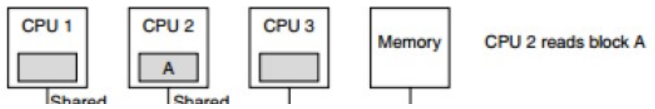
Exclusive: no other cache holds the line; memory is up to date.

Shared: multiple caches may hold the line; memory is up to date.

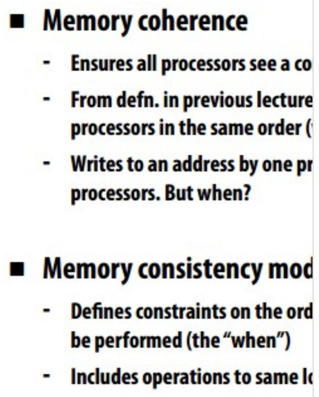
Invalid: the cache entry does not contain valid data

        - Example
          (read the operation on the right first)

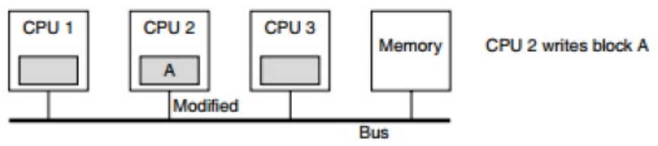




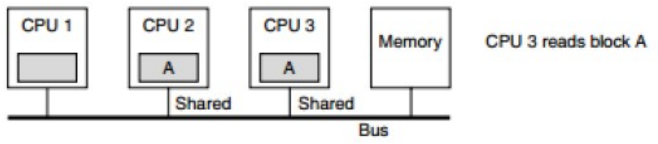
https://goo.gl/tvez9g

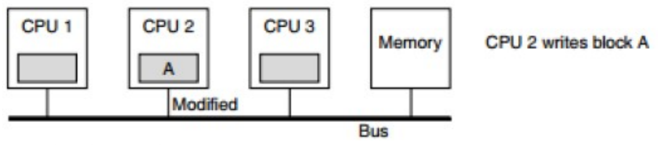


- **Memory coherence**
  - **Ensures all processors see a co**
  - **From defn. in previous lecture processors in the same order (**
  - **Writes to an address by one pr processors. But when?**
- **Memory consistency mod**
  - **Defines constraints on the ord be performed (the "when")**
  - **Includes operations to same l**



The CPU1's snooper snoop the write operation on the shared internal CPU bus and sets its cache block to the *invalid* state: for this reason, **MESI is an write-invalidate protocol** (with write-back caches).

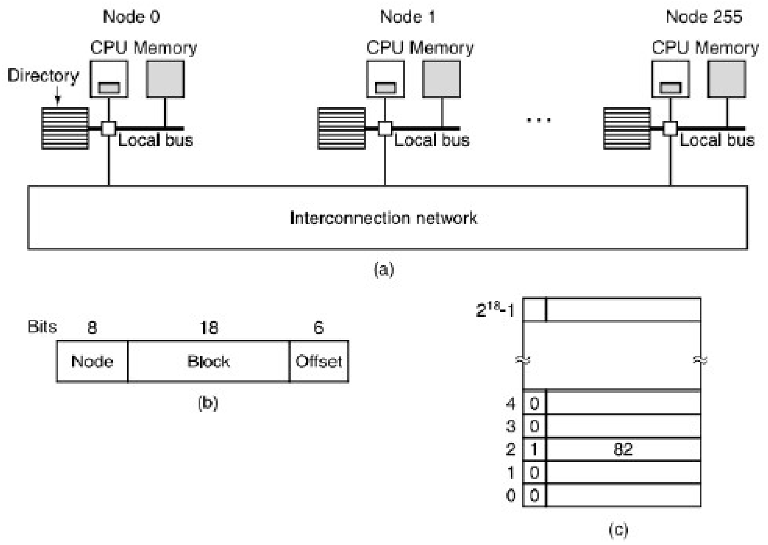




Same thing happens here again.

! • With/without intervention: not always a processor having a cache block in the **E**-state has to propagate the block when another processor asks for it.
When it happens, it's called MESI with intervention (or Extended-MESI).
When this doesn't happen (and the other processor has to get that cache block from main memory) it's called MESI without intervention.

- **Directory**-based protocols
  - Used in a **shared main memory** architectures (NUMA architecture).
  - Characteristics
  - ☐ ? Scheme (see it on the book)



  - ☐ ? Example
    - Let the local memory be 1MB.
    - Memory 0 contains the first MB, and so on.
    - CPU0 wants to access a certain address, divided in 3 parts:
      - Node
        - This is why subsequent memory locations are contained in one node.
      - Block
      - Offset

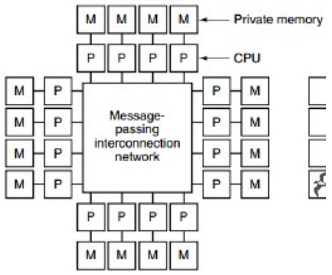
    - Figure (c) is the directory.
      The first bit indicates the memory block location:
      - 0: main memory
        A request to the system bus is then sent.
      - 1: another CPU's memory
        A request to the local bus is then sent.

      The other entry contains the CPU ID that contains the latest version of the desired memory block.

This architecture can support up to 100 processors.
The two rules to obtain good performances are:
  i. The workload b/w CPUs has to be balanced
  ii. The communication overhead has to be minimized
     1. Two processors that communicate very often should be very close

There are two solutions for this:

i. Distributed Shared Memory Machine (multi-computer)
(a node inside this architecture, a "P" in the image below, can be a multi-core processor)



The shared informations can be obtained and modified by mean of those operations:

1. Load
2. Store

? And…

1. Send
2. Receive

The directory is a shared structure that has, for each memory block in the system, the following informations:

- Its current state
    - There exist only one copy
    - There exist more copies
- The list of processor (nodes) that currently have a copy of the memory block

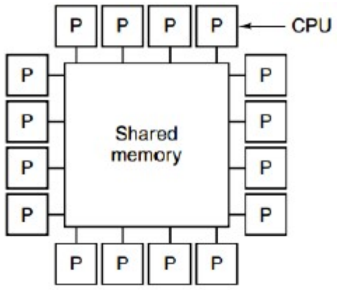
How is the write operation managed?

- Write-update
- Write-invalidate

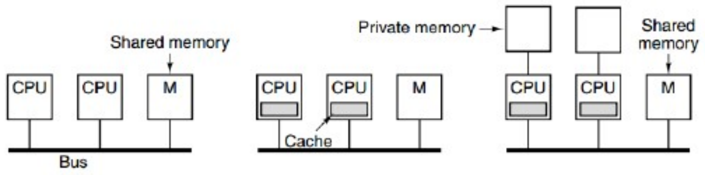
MultiProcessor Architectures

mercoledì 26 aprile 2017 09:49

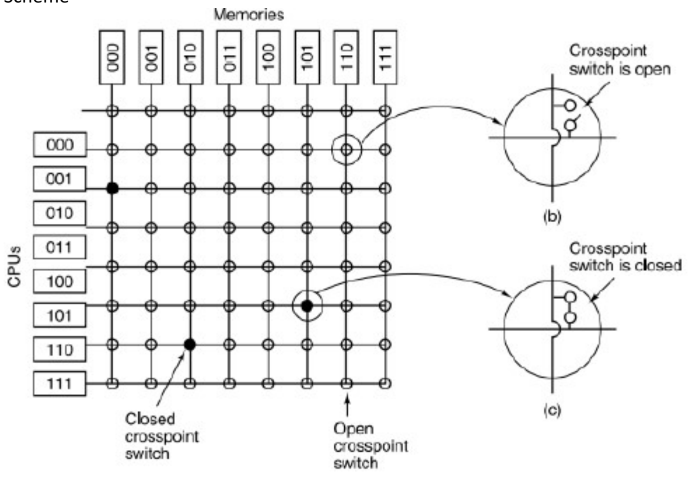
- Parallel computer architectures
    - SISD
    - SIMD
    - MISD
    - MIMD
        - Multi-processors
            - Scheme



            - Load and store instructions for communication b/w processors
            - If there are no special CPUs, all of them can be considered equal
            - Pros
                - Easy to program
                - Single shared copy of the O.S.
                  The O.S. knows how to balance the workload among all processors.
            - Different types
                - UMA
                    - Three implementations

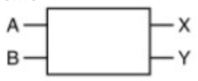


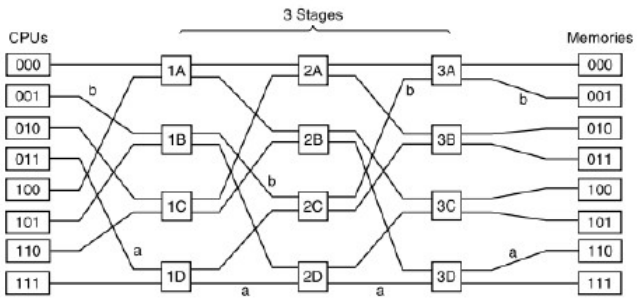
                      In the third one, CPUs have also a private memory.
                    - Interconnection network architectures
                        - Crossbar interconnection
                            - Scheme



                            - Pros
                                - There is not competition if different CPUs want to access to different memory locations.
                            - Cons
                                - High number of connections: n*k connections with n processors and k memories
                                - 2^(nk) open/closed combinations
                                - The maximum number of combinations that doesn't create any interferences are n! / (n*(n-k)!)
                        - Omega switching network

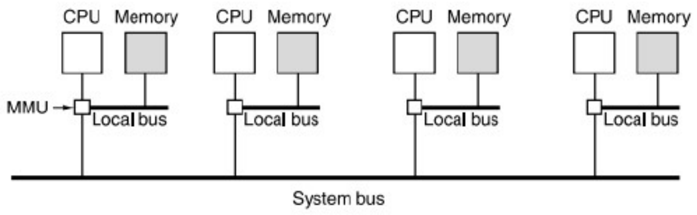
- Scheme



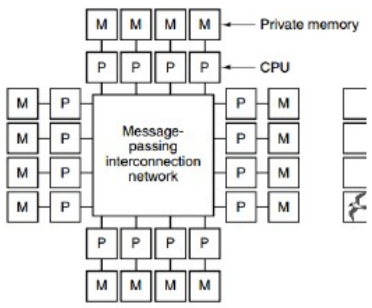


- ? How it works (see on the book, structured computer organization, chapter 8.3)
  - Each networks layer tests one CPU's ID bit, starting from the most significant one.
- Complexity
  - With n CPUs, $log_2 n \leq n^2$ layers are required.

- ~~COMA~~
- NUMA
  - Characteristics
    - The memory is not **private** (**it is both distributed and shared**) for each processor: there is a single address space visible to all CPUs.

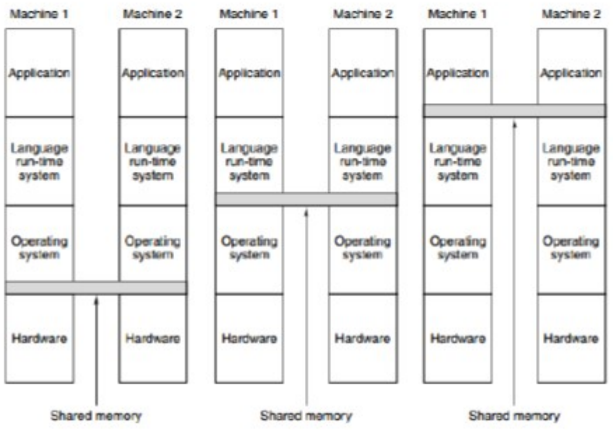


    - Access to remote memory is done using LOAD and STORE instructions. Those two operations are implemented in hardware.
    - Access to remote memory is slower than acccess to local memory.
      - Optimal for applications which reads their own memory portion most of the times.
    - Main problem: the execution time is not predictable at priori and it is variable.
    - It uses a directory-based cache coherency protocol.
- Multi-computers
  - Scheme



  - Send and receive instructions for communication b/w computers
  - Multiple copies of O.S.s
  - More page tables and more process tables
    Each CPU can then know which CPU is running a certain process.
  - Pros
    - Easy to build
      Just a bunch of computers connected over the internet.
  - Cons
    - Hard to program
      It's not easy to balance the workload among different computers
- Hybrid systems
  - Scheme

Hybrid System



- Load and store instructions are not performed via hardware, but via software over the interconnection network.