UT 3. Memory subsystem

Tema 3.2 Cache Performance Optimizations

A. Doménech, J. Duato, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Department of Computer Engineering
Universitat Politècnica de València

Contents

Bibliography

John L. Hennessy and David A. Patterson.
Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.

Contents

Introduction

Average access time

$T_{\text{access}} = Ht + MR \times MP$

where Ht: cache hit time
MR: miss rate
MP: miss penalty

How can we improve cache performance?
Reducing any term:

- Miss penalty (*MP*).
- Miss rate (*MR*).
- Cache hit time (*Ht*).

Contents

Techniques

- Multilevel caches.
- *Critical word first* and *Early restart*.
- Combined write buffers.

Multilevel caches

- Another cache level (L2) is added between the cache (first-level) and main memory:
  - The L1 cache is designed for speed, e.g., as fast as the processor.
  - The L2 cache is large to capture most of main memory accesses.
- How does it affect the access time equation?

$$T_{\text{access}} = Ht_{L1} + MR_{L1} \times MP_{L1}$$

The L1 cache miss penalty is:

$$MP_{L1} = Ht_{L2} + MR_{L2} \times MP_{L2}$$

Thus:

$$T_{\text{access}} = Ht_{L1} + MR_{L1} \times (Ht_{L2} + MR_{L2} \times MP_{L2})$$

Multilevel caches (cont.)

- With multiple cache levels there are two types of miss rate:

  Local miss rate = $\frac{\text{Num. cache misses}}{\text{Total cache accesses}}$

  - For two levels: MR_{L1} and MR_{L2}

  Global miss rate = $\frac{\text{Num. cache misses}}{\text{Total accesses}}$

  - For the L1 cache is MR_{L1}
  - For the L2 cache,

    $$\frac{\text{L2 misses}}{\text{Total accesses}} = \frac{\text{L1 misses}}{\text{Total accesses}} \cdot \frac{\text{L2 misses}}{\text{L1 misses}} = \frac{\text{L1 misses}}{\text{L1accesses}} \cdot \frac{\text{L2 misses}}{\text{L2 accesses}} = MR_{L1} \times MR_{L2}$$

  $\rightarrow$ MR $global_{L2} = MR_{L1} \cdot MR_{L2}$, i.e., fraction of accesses that reach main memory.

- From the miss rate point of view:
  - The global miss rate of a two-level cache system is similar to the one for a single-level cache as large as the L2
  - ...but with a smaller L1 cache $\rightarrow$ faster than the L2 alone.

Multilevel caches (cont.)

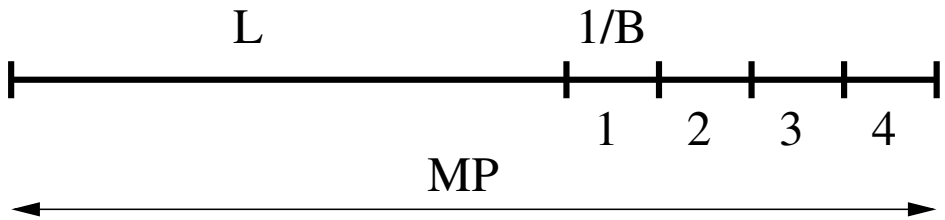
- Some design issues:
  - The L1 cache speed affects the processor clock cycle
    How do we reduce Ht_{L1}? (see slide 42)
    - Small L1 cache and direct-mapped or set-associative with few ways.
  - The L2 cache affects L1 miss penalty
    How do we reduce $MP_{L1} = Ht_{L2} + MR_{L2} \times MP_{L2}$?
    The second term is the dominant one as it multiplies the $MP_{L2} \rightarrow$
    Reduce MR_{L2} (see slide 20):
    - Large L2 cache, much larger than L1, and with many more ways.

Multilevel caches (cont.)

- Multilevel inclusion/exclusion
  - Multilevel inclusion
    - All the data in L1 cache are always in L2 cache.
    - Helps to maintain coherence $\rightarrow$ it is only necessary to check the lower level (L2).
    - In some processors the L2 block size is larger than in L1, or the L3 with respect to the L2. In case of replacement at the lower level, e.g. L2, all the blocks in the upper level (L1) that compose the L2 block must be invalidated, increasing MR_{L1}.
  - Multilevel exclusion (AMD Opteron)
    - There are not any replicated blocks in caches: located either in L1 or L2 $\rightarrow$ Better space management.
    - An L1 miss and L2 hit exchanges blocks between caches
    - Interesting when the L2 cache is only slightly larger than the L1 cache.
    - Coherence mechanism implementation is complex.

Critical word first and *Early restart*

The miss penalty *MP* depends on the memory latency *L* and bandwidth *B*



$MP = L + \frac{1}{B} \cdot n$, where n is the block size.

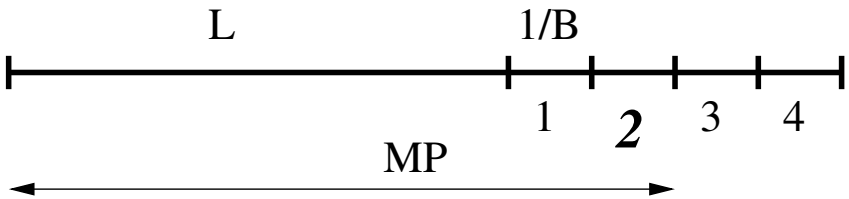
- Latency *L* or access time. Time elapsed since the address is sent to the memory until the first data is available at the memory output.
- Bandwidth *B*. Speed at which data can be supplied.

Critical word first and *Early restart* (cont.)

The processor only needs the word of the block that caused the miss.
$\rightarrow$ Why waiting to have the whole block loaded before delivering the word to the processor?

Solutions:

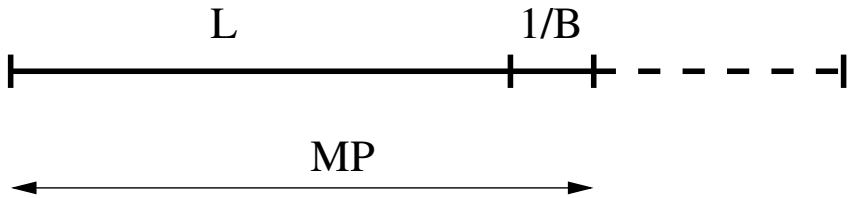
Early restart:



- Access words in the memory blocks as usual.
- As soon as the requested word arrives, it is delivered to the processor to continue execution.

Critical word first and *Early restart* (cont.)

Critical word first:



- The first accessed word is the one requested by the processor.
- The processor continues processing while the rest of the words in the block are loaded.

Performance improvement achieved with *Critical word first* and *Early restart* is larger when:

- Large cache blocks are used.
- The next memory instruction [1] *does not access* the same block that is being loaded $\rightarrow$ in such a case, the second access must wait till block loading finishes.

[1] Assuming non-blocking cache

Write *buffers*

Problem with writes: Memory is much slower than the processor → writes may stall the processor.

- When to write to memory?

  Write-through: Whenever the cache is written.
  Write-back: When a dirty block is replaced.

- Writes are not "producer" instructions. → they should not stall the processor.

Solution: write buffer

- The processor writes into the buffer without stalling execution, decoupling the execution from the memory write, which is handled by the controller.

Write *buffers* (cont.)

- Problem: memory data dependencies.

```
1 sd r3,a(r10)  ;  Write with miss to Mem[a+r10]
                ;  Write-through; no-write allocate
2 ld r1,b(r11)  ;
3 ld r2,a(r10)  ;  Read with miss from Mem[a+r10]
```

  $\rightarrow$ If the write from r3 (instr 1) has not been completed when instr 3 reads from memory, the loaded value would not be correct.

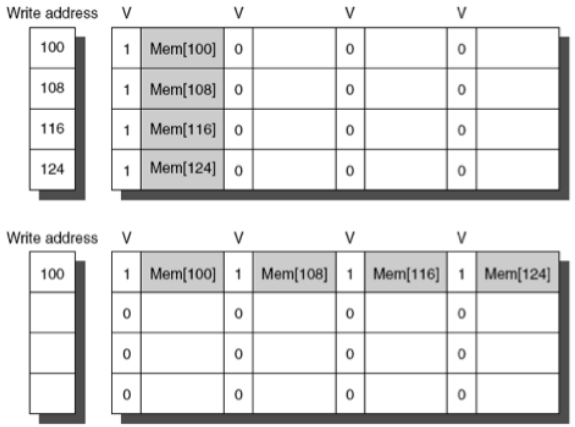
- Solutions:
  - Wait until the write buffer is emptied before reading data.
  - Check if the requested address is in the write buffer and, if not, let the read continue (*load-bypassing*).
  - If the requested address is in the write buffer, read the data from the buffer (*load-forwarding*).

Combined write *buffers*

If the buffer is full and an entry is needed $\rightarrow$ *stalls*.

Solution: Combined write buffers

- Each buffer entry refers to a set of consecutive addresses.
- The address is compared with those of valid buffer entries. If it matches, the data are combined with that entry.



Entries at the right of the upper figure are only used by multiword instructions.

Combined write *buffers* (cont.)

- Efficient memory bandwidth usage: Write several words at a time instead of only one.
- The Sun T1 (Niagara) and the Intel Core i7, among others, use combined writes to obtain a fast L1 to L2 *Write-through*.

Contents

Block miss classification

Compulsory Appear the first time a block is accessed.
→ Usually represent a low percentage of the total misses.

Conflict Appear when the target set is full but there is free space in other sets
→ Fully-associative caches do not have conflict misses, but require more hardware and may impact the clock frequency.

Capacity If the cache cannot store the entire working set for an application, active blocks are frequently replaced, thus producing capacity misses.
→ These misses are reduced when cache size increases.

The techniques to reduce the miss rate are intended to reduce the global miss rate.

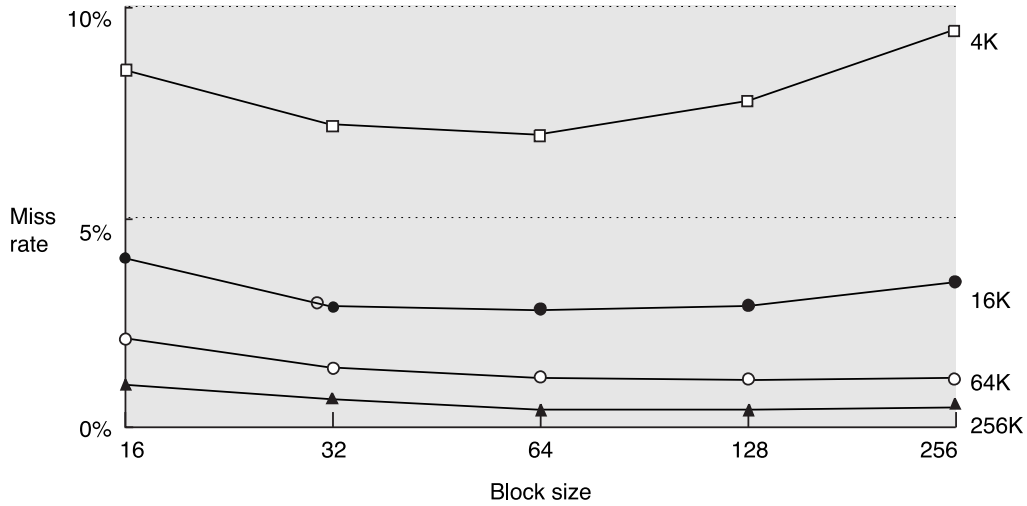
Techniques

- Adjusting the cache geometry:
  - Block size.
  - Cache size.
  - Number of ways.
- Way-prediction.
- Compiler optimizations.

Block size

A larger block size leads to:

- Higher exploitation of spatial locality $\rightarrow$ compulsory misses are reduced ($\downarrow$ *MR*)
- Less cache lines for a given cache size.
  - Conflict misses may rise ($\uparrow$ *MR*).
  - Capacity misses may rise ($\uparrow$ *MR*).



Block size (cont.)

- Increasing the block size increases the miss penalty ($\uparrow MP$) due to having to bring more words.
- How large should the block size be?
  Remember that $T_{\text{acc}} = Ht + MR \times MP$
  - The block size must be a tradeoff between MR and MP.
    Must be relatively large to amortize MP, but
    the miss rate must be reasonable.
- The typical size is 64B.
  Some processors use a larger size in the Last-Level Cache (LLC).
  For instance: L3 block size in the IBM POWER7 is 4 times larger than that of the L2.

Cache size

A larger cache size...

- Reduces capacity misses ($\downarrow MR$).
- Increases the hit time ($\uparrow Ht$).
- Increases area and power consumption.
- Idea: design L1 small with fast technology for performance, and L2 large for capacity, and
  use a power-aware technology (e.g. eDRAM) for huge caches.

Typically, the L1 size is 16KB or 32KB and LLCs are very large (tens of MB) to provide a good tradeoff between MR and MP.

Number of ways

A higher number of ways...

- Reduces conflict misses ($\downarrow MR$).
- Requires more comparators (more energy consumption).
- The multiplexer has more entries
  $\rightarrow$ may increase the cache access time ($\uparrow Ht$)

L1 caches have from 2 up to 8 ways, while LLCs sometimes exceed 20 ways.

Way-prediction

Objetives:

- Reduce conflict misses without increasing Ht with respect to a direct-mapped.
- Reduce energy with respect to a set-associative cache.

Idea: Way-prediction (Alpha 21264):

- The cache is set-associative.
- A predictor predicts the way of the set that has to be accessed. $\rightarrow$ multiplexer control entries are accordingly set in advance to hide multiplexer latency, while tag and data arrays are being accessed.
- On misprediction, the remaining tags are compared.

Way-prediction (cont.)

- Hit time has two different values:
  - **Prediction OK:** only one tag is compared → low Ht (1 cycle).
  - **Misprediction:** the remaining tags are compared, the predictor updated and the multiplexer properly set → Ht is much higher (3 cycles).

The prediction hits 85% of the times.
The behavior is similar to the one for a two-level cache.

$$T_{\text{access}} = \%\ \textit{hits} \times Ht_{\text{pred. hit}} + \%\ \textit{misses} \times Ht_{\text{misspred.}}$$

For the example:

$$T_{\text{access}} = 0.85 \times 1 + 0.15 \times 3$$

Compiler optimizations

The compiler generates an optimized code that reduces the miss rate.

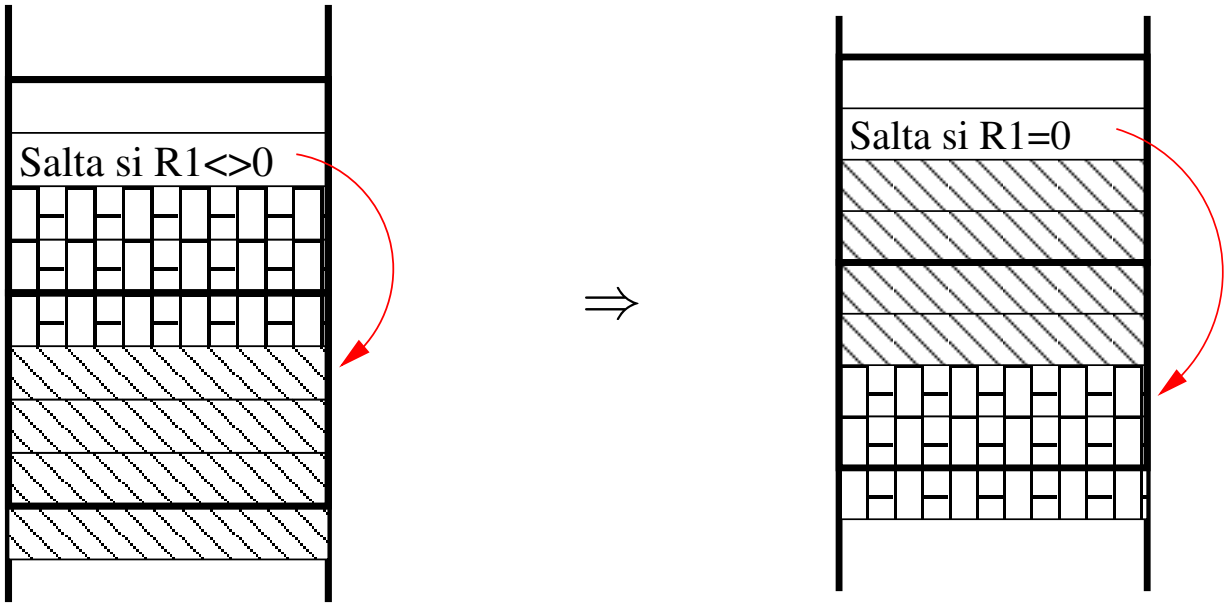
Reduction of instructions miss rate:

- Reordering the groups of instructions so that they are stored in different sets to reduce conflict misses.
- Aligning the entry point of basic blocks with the beginning of a cache block $\rightarrow$ improves spatial locality.

Compiler optimizations (cont.)

- *Branch straightening*. If the compiler predicts a branch to be taken, the code is modified to:
  - Evaluate the opposite condition, and
  - Locate next to the branch instruction the code initially located at the target address

$\rightarrow$ improves spatial locality.



Compiler optimizations (cont.)

Reduction of data misses:

Example: matrix operations. Code reordering in order to operate over all data in a block before requesting data from the next block.

Improving the spatial locality Loop exchange.

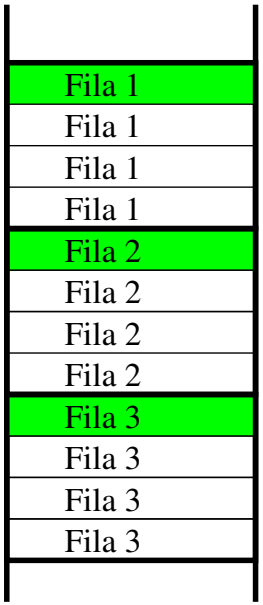
Loop nesting is exchanged to operate in the order in which data are stored in memory.

Compiler optimizations (cont.)

Example, matrix x stored in row-major order:

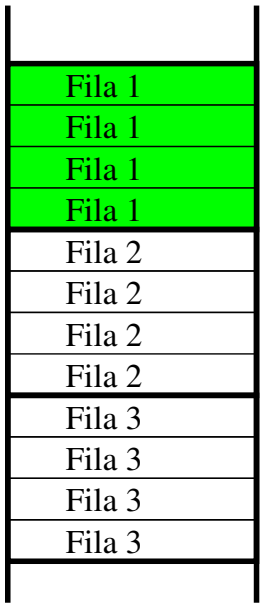
```
/* before */
for (j=0; j<100; j=j+1)
   for (i=0; i<5000; i=i+1)
      x[i][j] = 2* x[i][j]
```

→ accesses with 100-word strides




```
/* after */
for (i=0; i<5000; i=i+1)
   for (j=0; j<100; j=j+1)
      x[i][j] = 2* x[i][j]
```

→ words sequentially accessed



Compiler optimizations (cont.)

Improving the temporal locality: *blocking*

If arrays are accessed both by rows and by columns, it is better to operate on sub-matrices or blocks, trying to maximize the access to cached data before lines are replaced.

Example: matrix multiplication

```
/* before */
for (i=0; i<N; i++)
  for (j=0; j<N; j++) {
    r=0;
    for (k=0; k<N; k++)
      r= r+y[i][k]*z[k][j]
    x[i][j]= r;
  }
```

```
/* after */
for (jj=0; jj<N; jj=jj+B)
  for (kk=0; kk<N; kk=kk+B)
    for (i=0; i<N; i++)
      for (j=jj; j< min(jj+B,N); j++){
        r=0;
        for (k=kk; k<min(kk+B,N); k++)
          r= r+y[i][k]*z[k][j]
        x[i][j]= x[i][j]+r;
      }
```

Contents

Concepts and techniques

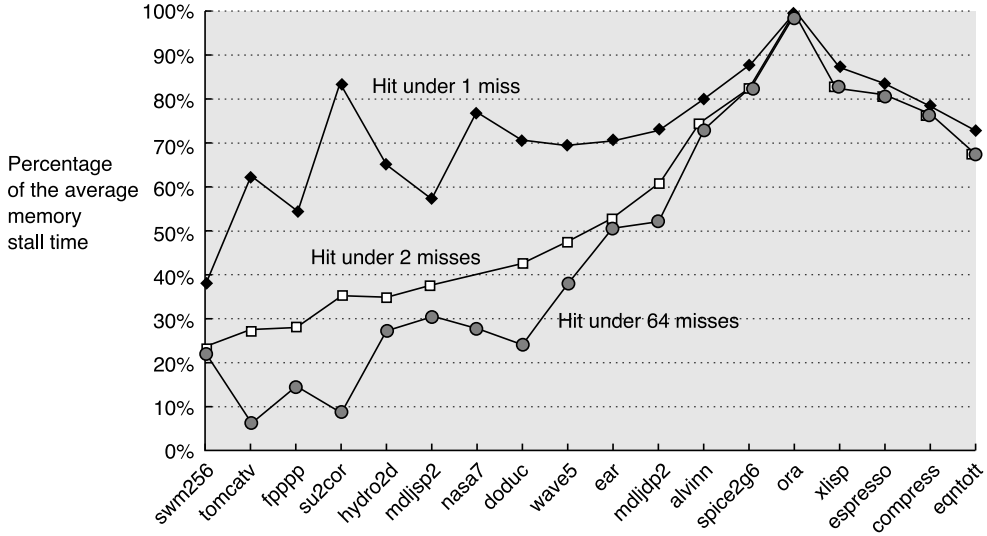
→ technique that overlaps execution of instructions with memory access

- Non-blocking caches
- Hardware-based prefetching of instructions and data
- Compiler-controlled prefetching

Non-blocking caches

- The cache services new requests while a previous miss is being handled
- Alternatives:
  - “hit under miss”: the cache can only handle one miss at a time, but in the meantime, it is able to service hits
  - “miss under miss” or “hit under multiple misses”: multiple misses can be handled at the same time

Non-blocking caches (cont.)



Hardware-based prefetching of instructions and data

Idea: look for information before it is required by the processor

- Pre-fetched information is stored in an external buffer, faster than main memory
- Prefetch of instructions
  - On a miss, the processor retrieves two blocks: the required one, which is stored in the cache, and the following one, which is stored in the instructions buffer
  - On a cache miss, the block in the instructions buffer is read, and in case of hit, the next block is prefetched
  - Evaluation of the proposal.
    Cache: 4KB size, 16B blocks
    Misses avoided: A single buffer avoids 15%–25%; 4 buffers avoid 50%, and 16 buffers avoid 72% of the misses.

Hardware-based prefetching of instructions and data (cont.)

- Data prefetching
  - More complex than instruction prefetching
  - The block to be prefetched is computed by the prefetcher logic. Most current prefetchers detect non-unit regular stride patterns
- Prefetch requests compete with cache misses when accessing to main memory $\rightarrow$ MP may increase

Compiler-controlled prefetching

- The compiler inserts prefetch instructions in order to request data before the processor requires them.
- Prefetch instructions must not generate neither virtual memory faults nor protection violation exceptions $\rightarrow$ *nonbinding fetch*
- It makes sense in non-blocking caches
- Especially useful in loops. Example:
  Cache: 8KB, 16B line, direct-mapped. Write-back and no-write allocate
  a is 3x100 array and b is 101x3 array, both of floating-point (8B) numbers
  Assume that the MP is so high that 7 iterations are prefetched in advance

  First accesses are not prefetched and last prefetches are not eliminated

Tema 3.2 Cache Performance Optimizations

```
/* original */
for (i=0; i<3; i=i+1)
  for (j=0; j<100; j=j+1)
      a[i][j] = b[j][0]*
                b[j+1][0];
```

Total misses: 150(a)+101(b)=251

```
/* with prefetch */
for (j=0; j<100; j=j+1) {
/* b[j,0] for 7 iterations later */
  prefetch(b[j+7][0])
/* a[0,j] for 7 it. later */
  prefetch(a[0][j+7])
  a[0][j] = b[j][0]*
            b[j+1][0];};

for (i=1; i<3; i=i+1)
  for (j=0; j<100; j=j+1) {
/* a[i,j] for 7 it. later  */
      prefetch(a[i][j+7])
      a[i][j] = b[j][0]*
                b[j+1][0];};
```

Total misses: 19; Extra instructions: 400

- Overhead: insertion of prefetch instructions increases the # of executed instructions → The focus must be placed on those accesses likely to result in block misses

Contents

Concepts and techniques

Reducing hit time is very important: it affects the processor clock period (i.e. T variable in the execution time equation).

- Simple and small caches
- Avoid virtual memory translations in cache accesses
- Cache pipelining

Simple and small caches

- A large % of the time required to access the cache is devoted to compare the tag field of the accessed address with the tags stored in the cache
- A possible way to reduce this time:

  Small cache: "the smaller the cache structure the faster" and it fits in the same chip the processor does (L1 and L2)

  Simple cache: direct mapped caches (this way the multiplexer is not required), or set-associative with few ways

  - Direct-mapped caches are 1.2-1.5 times faster than 2-way (associative) ones
  - 2-way caches are 1.02-1.11 faster than 4-way ones
  - 4-way associative caches are 1-1.08 faster than 8-way ones

Simple and small caches (cont.)

- Trends:

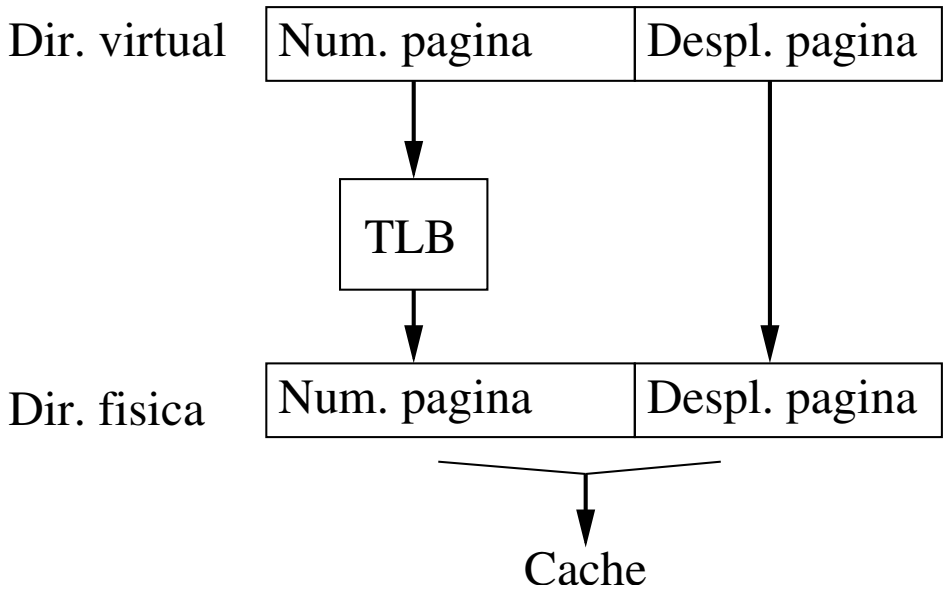
  L1 caches: small and simple. Emphasis on speed for performance.
  - Reduced size and low associativity.
  - A lot of L1 caches are 2-way associative although some cores include 8-way caches.

  L2 cache: much larger to avoid memory accesses.
  - They are designed to be private (accessed by just one core) or shared by a subset of the cores.

  L3 cache: some processors include L3 as LLC.
  - These caches are shared by several L2 caches.
  - Designed to avoid accesses to main memory. In particular, they are useful to store shared variables.
  - These caches are huge and use to have more than 20 ways.
  - Low-power technologies are used to implement them.

Avoid virtual memory translations in cache accesses

Another component of the Ht is the time devoted to translate from the virtual memory address issued by the processor to the physical memory address.
This translation is performed in the TLB. Once the translation completes, the cache access can be performed



Avoid virtual memory translations in cache accesses (cont.)

- Idea. Use the virtual address to access the cache.
  Virtual Caches vs. Physical caches
- Problems:

  Protection This is part of the translation process of the virtual address to the physical one $\rightarrow$ Information from the TLB must be copied to the cache.

  Processes Each process owns its virtual memory space. When the context is switched, a given virtual address points to a different physical address
  $\rightarrow$ the cache must be emptied with each context switch or process identifiers (PID) must be added to cache tags

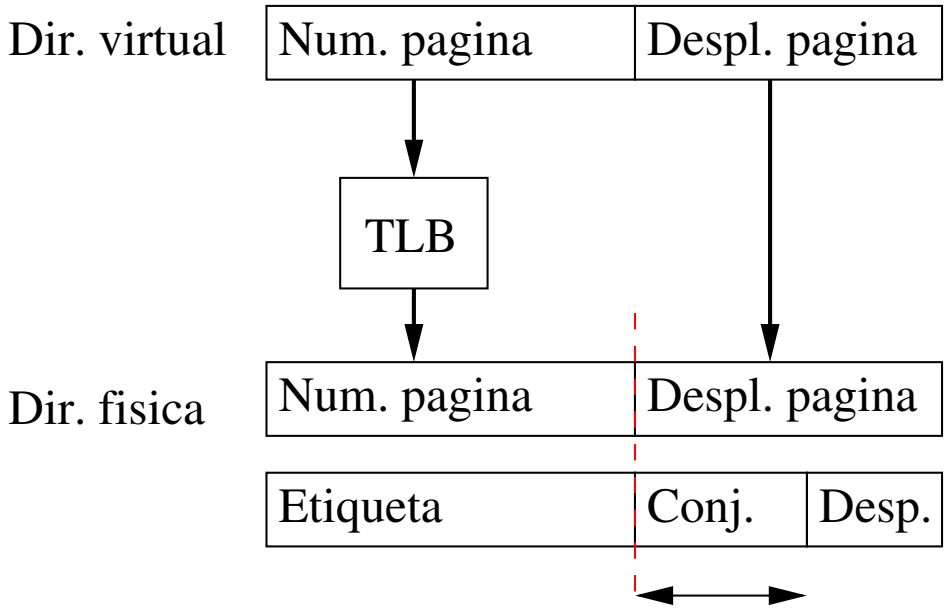
Avoid virtual memory translations in cache accesses (cont.)

Aliasing A given physical address can be referred to by using two or more different virtual addresses.
$\rightarrow$ There may exist several copies of the same data that must be kept consistent (they must be identical).

- *Virtually indexed physically tagged caches* (Alpha 21264).

Observation: The page offset (address within a page) is the same in both virtual and physical addresses.

Avoid virtual memory translations in cache accesses (cont.)



- A part of the page offset is used to index the cache.
- Cache tags and data are read in parallel to the translation.
- Limitation:
  The size of a direct-mapped cache or the number of sets in a set-associative cache cannot exceed the size of a virtual memory page.

Cache pipelining

- Cache access is pipelined in stages to match the processor clock.
- A cache access takes several cycles (for instance, 4 cycles in the Pentium 4) but several instructions can overlap their cache accesses.
- It impacts the design of the processor pipeline: more stall cycles in branch and load instructions

This technique increases the instruction cache bandwidth rather than reducing its latency