Cache performance
Outline

- Metrics
- Performance characterization
- Cache optimization techniques
Cache Performance metrics (1)

- **Miss rate:**
  - *Neglects* cycle time implications

- **Average memory access time** (AMAT):
  \[ AMAT = \text{Hit time} + (\text{Miss Rate} \times \text{Miss Penalty}) \]
  - *miss penalty* is the extra time it takes to handle a miss (above the 1 cycle hit cost)

- Example:
  - 1 cycle hit cost
  - 10 cycle miss penalty (11 cycles total for a miss)
  - Program has 10% miss rate
  - Average memory access time = 1.0 + 10% * 10 = 2.0
Cache performance: example

- Two cache options:
  a) 16 KB instruction + 16 KB data
  b) 32 KB unified cache (instructions + data)

- Information:
  - Hit = 1 cycle, miss = 50 cycles
  - Load/store = 1 extra cycle on the unified cache (single-port)
  - 75% of memory references are fetches
  - Miss rate info:

<table>
<thead>
<tr>
<th>Cache size</th>
<th>I-cache</th>
<th>D-cache</th>
<th>Unified</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB</td>
<td>3.0 %</td>
<td>24.6%</td>
<td>13.3%</td>
</tr>
<tr>
<td>2KB</td>
<td>2.3%</td>
<td>20.6%</td>
<td>9.8%</td>
</tr>
<tr>
<td>4KB</td>
<td>1.8%</td>
<td>16.0%</td>
<td>7.2%</td>
</tr>
<tr>
<td>8KB</td>
<td>1.1%</td>
<td>10.2%</td>
<td>4.6%</td>
</tr>
<tr>
<td>16KB</td>
<td>0.6%</td>
<td>6.5%</td>
<td>2.9%</td>
</tr>
<tr>
<td>32KB</td>
<td>0.4%</td>
<td>4.8%</td>
<td>2.0%</td>
</tr>
<tr>
<td>64KB</td>
<td>0.1%</td>
<td>3.8%</td>
<td>1.3%</td>
</tr>
</tbody>
</table>
Cache performance: example (2)

- **Comparing miss rates**
  a) **Split cache:**
    - Miss rate instruction (16KB) = 0.6%
    - Miss rate data (16KB) = 6.5%
    - Overall miss rate = 0.75*0.6 + 0.25*6.5 = 2.07
  b) **Unified cache**
    - Miss rate (32KB) = 2.00

- **Comparing cycle time penalty**
  a) \( \text{AMAT} = 0.75* (1 + 0.6\%*50) + 0.25* (1 + 6.5\%*50) = 2.05 \)
  b) \( \text{AMAT} = 0.75* (1 + 0.6\%*50) + 0.25* (1 + 1 + 6.5\%*50) = 2.24 \)
Cache Performance metrics (2)

- **CPU time**: the ultimate metric
  
  - \( CPU \) time = execution + memory access
  
  \[
  CPU \text{ time} = (CPU \text{ execution cycles} + Memory \text{ stall cycles}) \times T_{cycle}
  \]

  - Assuming that all memory stalls are due to caches:
    
    Memory stall cycles =
    
    \# Reads \times Read miss rate \times Read miss penalty +
    
    \# Writes \times Write miss rate \times Write miss penalty

  - Combining reads and writes together:
    
    Memory stall cycles =
    
    **Memory accesses \times Miss rate \times Miss penalty**
Cache Performance metrics (3)

- Factoring instruction count IC

\[
\text{CPU}_{\text{time}} = IC \times (\text{CPI}_{\text{exec}} + \text{Mem. accesses per instruction} \times \text{Miss rate} \times \text{Miss penalty}) \times T_{\text{cycle}}
\]

- Measuring miss rate as misses per instruction

Misses per instruction = Memory accesses per instruction \times Miss rate

\[
\text{CPU}_{\text{time}} = IC \times (\text{CPI}_{\text{exec}} + \text{Misses per instruction} \times \text{Miss penalty}) \times T_{\text{cycle}}
\]
Cache Performance metrics

◆ Summarizing:

◆ Miss rates

◆ AMAT

\[ \text{Hit time} + (\text{Miss Rate} \times \text{Miss Penalty}) \]

◆ CPU time

\[ \text{IC} \times (\text{CPI}_{\text{exec}} + \text{Misses per instruction} \times \text{Miss penalty}) \times T_{\text{cycle}} \]
**Improving Cache Performance**

\[
\text{AMAT} = \text{Hit time} + (\text{Miss Rate} \times \text{Miss Penalty})
\]

1. Reduce the miss rate
2. Reduce the miss penalty
3. Reduce the time to hit in the cache
Reducing misses
Classifying Misses: The 3 C’s

- **Compulsory**:  
  - The first access to a block is not in the cache, so the block must be brought into the cache. Also called *cold start misses* or *first reference misses*.  
  - *Misses even in an Infinite Cache*

- **Capacity**:  
  - Will occur due to blocks being discarded and later retrieved, if the cache cannot contain all the blocks needed during execution of a program.

- **Conflict**:  
  - Occur in set associative or direct mapped caches because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses*.  

- **Invalidation**:  
  - Other process (e.g., I/O) updates memory.
3Cs Absolute Miss Rate (SPEC92)
2:1 Cache Rule

miss rate 1-way associative cache size $X$ 
≈ miss rate 2-way associative cache size $X/2$
3Cs Relative Miss Rate

Flaws: for fixed block size
How Can we Reduce Misses?

◆ Intuitively:
  ◆ Capacity misses are reduced by *increasing cache size*
  ◆ Conflict misses are reduced by *increasing associativity*
  ◆ Compulsory misses cannot be reduced
    • Except by prefetching (achieved with larger block sizes)

◆ Specific optimizations:
  1. Larger block sizes
  2. Higher associativity
  3. Smart indexing
  4. Victim caches
  5. HW prefetching
  6. Compiler optimizations
Larger Block Sizes (1)

- Larger blocks reduce the number of compulsory misses because more data is brought into the cache on each miss
  - High degree of spatial locality!
- Drawbacks:
  - Larger blocks eventually create cache pollution
    - More data is evicted from the cache
  - Larger block sizes increase the miss penalty (and thus AMAT)
    - Larger blocks take more cycles to refill
- There exists an optimal block size
Larger Block Sizes (2)

![Graph showing miss rate against block size for different block sizes: 1K, 4K, 16K, 64K, and 256K. The x-axis represents block size in bytes, ranging from 16 to 256, and the y-axis represents miss rate percentages from 0% to 25%. The graph illustrates how miss rate increases with block size for each block size category.]

- 1K
- 4K
- 16K
- 64K
- 256K
Larger Block Sizes (3)

- Increasing block size increases AMAT!
  - Larger blocks require multiple “bus cycles” to be transferred
- Example:

<table>
<thead>
<tr>
<th>Block size</th>
<th>Miss penalty</th>
<th>1k</th>
<th>4k</th>
<th>16k</th>
<th>64k</th>
<th>256k</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>42</td>
<td>7.3</td>
<td>4.6</td>
<td>2.6</td>
<td>1.8</td>
<td>1.4</td>
</tr>
<tr>
<td>32</td>
<td>44</td>
<td><strong>6.8</strong></td>
<td><strong>4.2</strong></td>
<td><strong>2.2</strong></td>
<td>1.6</td>
<td>1.3</td>
</tr>
<tr>
<td>64</td>
<td>48</td>
<td>7.6</td>
<td>4.4</td>
<td>2.2</td>
<td><strong>1.5</strong></td>
<td><strong>1.2</strong></td>
</tr>
<tr>
<td>128</td>
<td>56</td>
<td>10.3</td>
<td>5.3</td>
<td>2.5</td>
<td>1.6</td>
<td>1.2</td>
</tr>
<tr>
<td>256</td>
<td>72</td>
<td>16.8</td>
<td>7.8</td>
<td>3.3</td>
<td>1.8</td>
<td>1.3</td>
</tr>
</tbody>
</table>
Higher Associativity

- Higher degrees of associativity **reduce conflict misses**
  - 8-way set-associative caches are almost as good as fully associative caches

- **2:1 Cache Rule:**
  - Miss Rate DM cache of size $N \approx$ Miss Rate 2-way cache of size $N/2$

- **Drawback:**
  - Increased associativity can increase the access time
    - More logic for compares
    - Set-associative caches not used as primary caches (where cycle-time is most important) but as lower-level caches (where access time is less of an issue)
# Higher Associativity (2)

- AMAT for different cache configurations:
  - Red values = AMAT does not decrease

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.33</td>
<td>2.15</td>
<td>2.07</td>
<td>2.01</td>
</tr>
<tr>
<td>2</td>
<td>1.98</td>
<td>1.86</td>
<td>1.76</td>
<td>1.68</td>
</tr>
<tr>
<td>4</td>
<td>1.72</td>
<td>1.67</td>
<td>1.61</td>
<td>1.53</td>
</tr>
<tr>
<td>8</td>
<td>1.46</td>
<td>1.48</td>
<td>1.47</td>
<td>1.43</td>
</tr>
<tr>
<td>16</td>
<td>1.29</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
</tr>
<tr>
<td>32</td>
<td>1.20</td>
<td>1.24</td>
<td>1.25</td>
<td>1.27</td>
</tr>
<tr>
<td>64</td>
<td>1.14</td>
<td>1.20</td>
<td>1.21</td>
<td>1.23</td>
</tr>
<tr>
<td>128</td>
<td>1.10</td>
<td>1.17</td>
<td>1.18</td>
<td>1.20</td>
</tr>
</tbody>
</table>
Smart indexing (1)

- Target: conflict misses
- Conventional indexing mechanism uses $m$ LS bits of address ($2^m$ lines)
  - Choice driven by performance (fast translation)
- Cache indexing is a hashing problem
  - Map $2^n$ to $2^m$ lines by minimizing conflicts
- Better indexing: use a hash function
  - Problem: good hash function are costly!
  - Find a “reasonable” hash function
Smart indexing (2)

• Options:
  • Use an alternative subset of address bits
    • Example \((n=4, m=2)\)
      
      
      \[
      \begin{array}{c|c|c|c|c|c|c}
        & 0000 & 0011 & 0111 & 0001 & 0101 & 1011 \\
        \hline
        0000 & \text{Conflict} & \text{No conflict} & \text{Conflict} & \text{No conflict} & \text{Conflict} & \text{No conflict} \\
      \end{array}
      \]

    • Same complexity as standard indexing, modest reductions

  • Low-cost hashing
    • Use XOR between selected bits
    • Bit \(k\) of index is \(b_k = a_i \otimes a_j \otimes \ldots \otimes a_p\) \(k=1,\ldots, m\) \(p \leq n\)
    • Cost of XORs can be amortized into address generation pipeline stage
Victim Cache

- [Jouppi 1990]
- Small cache that contains the most recently discarded (replaced) cache blocks
  - Typically 1-5 blocks
  - **On cache miss, victim cache is checked**
    - If block is present, victim cache block is placed back into the primary cache
- Equivalent to increased associativity for a few lines
- Very effective for direct-mapped caches
  - A four-entry victim cache can remove 20% to 95% of conflict misses in a 4-KByte direct-mapped cache
- Used in Alpha, HP machines
Hardware Prefetching of Instructions & Data

- On a cache miss, fetch both the missing block and subsequent blocks
  - Prefetched blocks are not placed directly into the cache, but into a special **prefetch buffer**
  - Avoids cache pollution
- Very effective for instruction streams, but works for data as well
- Equivalent to a bigger blocksize for one or a few of the cache lines.
- Used in the Alpha 21064/21164
  - 21064 fetches two cache blocks on a miss (2nd block is a prefetch)
Compiler Optimizations

Instructions:
- Make the *non-branch path* the common case
- Isolate all exception-handling code together

Data:
- **Array Merging**
- **Loop Interchange:**
  - change nesting of loops to access data in order stored in memory
- **Loop Fusion:**
  - Combine 2 independent loops that have same looping and some variables overlap
- **Blocking:**
  - Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows
Array Merging

/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of structures */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];

◆ Reduces conflicts between val & key;
   improve spatial locality
Loop Interchange

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

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

◆ Sequential accesses instead of striding through memory every 100 words
Loop Fusion

/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];

◆ 2 misses per access to a & c vs. one miss per access
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1){
        r = 0;
        for (k = 0; k < N; k = k+1){
            r = r + y[i][k]*z[k][j];
        }
        x[i][j] = r;
    }

◆ Two Inner Loops:
  ◆ Read all NxN elements of z[]
  ◆ Read N elements of 1 row of y[] repeatedly
  ◆ Write N elements of 1 row of x[]

◆ Capacity Misses are a function of N & Cache Size:
  ◆ 3 NxNx4 => no capacity misses; otherwise ...
  ◆ Idea: compute on BxB submatrix that fits
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
  for (j = jj; j < min(jj+B-1,N); j = j+1)
    {r = 0;
     for (k = kk; k < min(kk+B-1,N); k = k+1) {
       r = r + y[i][k]*z[k][j];
     }
    x[i][j] = x[i][j] + r;
  }

◆ B called **Blocking Factor**
◆ Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
Reducing Conflict Misses by Blocking

- Conflict misses in caches not FA vs. Blocking size
  - Lam et al [1991] a blocking factor of 24 had 1/5 the misses vs. 48 despite both fit in cache
Summary of Compiler Optimizations

- vpenta (nasa7)
- gmtty (nasa7)
- tomcatv
- btrix (nasa7)
- mxm (nasa7)
- spice
- cholesky (nasa7)
- compress

Performance Improvement

- merged arrays
- loop interchange
- loop fusion
- blocking
Reducing miss penalty
Reducing miss penalty

How to reduce the cost of a miss

- Obvious solution: Make DRAM faster

Four techniques:

1. Giving priority to reads over writes
2. Sub-block placement
3. Early restart
4. 2nd-level caches
Read Priority over Write on Miss

- Write-buffers allow reads to proceed while writes wait for idle bus cycles

- Possible problems:
  - Example:
    - Direct-mapped cache, 512 & 1024 mapped on same block
    - `SW 512(R0), R3`; assume cache index is 0
    - `LW R1,1024(R0)`; evicts data from previous store
    - `LW R2,512(R0)`; misses again
    
    This is a RAW hazard if the write is buffered in the write buffer

- Solution:
  - Let the write buffer empty on a read-miss (i.e., wait)
  - High penalty (for read miss)

- Alternative:
  - check write buffer contents before read;
    if no conflicts, let the memory access continue
Read Priority over Write on Miss (2)

✦ Write-back caches:
  ✦ Read miss replacing dirty block
  ✦ Normal:
    • Write dirty block to memory, and then do the read
  ✦ Alternative:
    • Copy the dirty block to a write buffer, then do the read, and then do the write
    • CPU stalls less since restarts as soon as read is done
Subblock Placement

- Don’t have to load full block on a miss
  - Divide a cache line into sub-blocks
  - Use one **valid bit** per subblock to denote if subblock is valid
    - Originally invented to reduce tag storage
- Only refill part of a line on a miss
- Only write-back the dirty part of a line sub-blocks

![Diagram showing subblock placement and valid bits](image-url)
Early Restart and Critical Word First

- Don’t wait for full block to be loaded before restarting CPU
  \[ t_{miss} = t_{access} + t_{transfer} \]

- Early restart:
  - As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution

- Critical Word First (wrapped fetch)
  - Request the missed word first from memory and send it to the CPU as soon as it arrives (requires bus & memory arrangement)
  - Let the CPU continue execution while filling the rest of the words in the block.

- Generally useful only in large blocks
- Spatial locality a problem; tend to want next sequential word, so not clear if benefit by early restart
Non-blocking Caches

- Non-blocking (or lockup-free) cache allows to handle hits while a miss is pending
  - Requires out-of-order execution CPU
- Classified according to number of allowed outstanding misses:
  - One outstanding miss ("hit under miss")
    - reduces the effective miss penalty by working during miss vs. ignoring CPU requests
  - Multiple outstanding misses ("hit under multiple miss" or "miss under miss")
    - Generalization
    - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
    - Requires multiple memory banks (otherwise cannot be supported)
    - Pentium Pro allows 4 outstanding memory misses
Non-blocking cache potential

Definitions:

- **Primary miss**: first miss to a main memory block
- **Secondary misses**: subsequent misses to the block being fetched
- **Structural-stall misses**: subsequent misses to the block being fetched that stall because of resource contention
Non-blocking cache (2)

- Additional HW required:
  - **Miss Status Holding Registers** (MSHRs)
    - Compared on each miss
    - Stalls reads to pending miss
    - Buffers writes to pending miss
    - Holds information on all outstanding misses
      - *Per block* data: status & pointer to frame
      - *Per word* data: destination register, overwritten, in-buffer bits
**Value of Hit Under Miss for SPEC**

Hit Under i Misses

- **FP programs on average:** AMAT = 0.68 -> 0.52 -> 0.34 -> 0.26
- **Int programs on average:** AMAT = 0.24 -> 0.20 -> 0.19 -> 0.19
- **8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss**
Second Level Cache

✶ L2 Equations

\[
\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}
\]

Miss Penalty_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}

\[
\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})
\]

✶ Definitions:

✧ *Local miss rate:*
  - Misses in this cache divided by the total number of memory accesses *to this cache* (Miss rate_{L2})

✧ *Global miss rate:*
  - misses in this cache divided by the total number of memory accesses *generated by the CPU* (Miss Rate_{L1} \times Miss Rate_{L2})

✧ **Global Miss Rate is what matters** (because of cost of miss)
Second-level cache

- The local miss rate of L2 caches typically quite bad
  - L1 has caught most locality!
- Primary target: low miss rate
  - Large size
  - Large block size
  - High associativity
  - Latency not a big issue
- Usually provide *multi-level inclusion*:
  - Everything at level i is also in level i+1
  - Not always necessary
Comparing Local and Global Miss Rates

- 32 KByte L1 cache
- X-axis = L2 cache size
- Global miss rate close to single level cache rate provided that $L_2 \gg L_1$
  - Local miss rate not relevant
  - Since hits are few, target miss reduction
Making RAM faster

- In general, we assume that fastest memory technology is used for DRAM

- Other options to speed up misses:
  - **Make the memory wider**
    - Ex: Read out all 2 (or more) words in parallel
    - Overhead:
      - Higher cost, wider bus, error-correction harder
  - **Interleave Main Memory**
    - Requires multibank memories
Reducing hit time
Reducing hit time

- Use simpler caches:
  - Use direct-mapped caches
    - No multiplexer in the path
    - Data can be speculatively used while tag check is in progress
  - **Write hits take longer** because tag must be checked before writing (read and write are in sequence).
  - To fix, **pipeline the writes**:
    WRITE1: readtag writecache
    WRITE2: readtag writecache

![Diagram of cache operations]

Tag mem  | R0  | R1  | R2  | W3  | W4  | R5  | R6  | W7  
Data mem | R0  | R1  | R2  | W3  | R5  | R6  | W4  | W7  
Write buffer | W3  | W4  | W4  | W4  | W7  

- tag
- data

mux

Write buffer

Tag mem

Data mem
Reducing hit time

- Pipelining writes
  1. Check tag for write hit
  2. Actually write data
- Need a 2\textsuperscript{nd} port for reads, otherwise stall for read after a write

(Hennessy & Patterson Figure 5.28)
# Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Miss rate</th>
<th>Miss penalty</th>
<th>Hit time</th>
<th>Hardware complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger block size</td>
<td>+</td>
<td>-</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Higher associativity</td>
<td>+</td>
<td>-</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Victim caches</td>
<td>+</td>
<td>-</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Pseudo-associative caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Hardware prefetching of instructions and data</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Compiler-controller prefetching</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Compiler techniques to reduce cache misses</td>
<td>+</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Giving priority to read misses over writes</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Subblock placement</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Early restart and critical word first</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Nonblocking caches</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Second-level caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Small and simple caches</td>
<td>-</td>
<td>+</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Avoiding address translation during indexing of the cache</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Pipelining writes for fast write hits</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Other cache issues
Outline

- Relation with virtual memory
- Example of real memory hierarchies
Virtual memory: review

- From OS theory: *Separation of virtual and physical address spaces*
  - **Virtual**: as seen by the program (CPU)
  - **Physical**: as seen by the memory

- Moving from virtual to physical requires a translation
  - For realistic memory management schemes, done by hardware (MMU)
    - Paging
    - Segmentation
  - Done for each memory access
Virtual memory (VM)

- Most systems use paging (sometimes combined with segmentation)
  - Virtual addresses (VA) = pages
  - Physical addresses (PA) = frames
- Translation: page table
  - Typically, stored in main memory
  - Entry contains
    - Physical frame number
    - Status bits (valid, dirty, protection...)
- Typically a special “cache” called Translation Lookaside Buffer (TLB) is used to speed up most common translations
  - 32 to 1024 entries (slots)
  - Can be any organization, but typically fully associative
VM and translation

(no TLB)

virtual address

31 12 11 0

virtual page number page offset

valid access physical page number

if valid=0 then page is not in memory and page fault exception

physical page number

29 12 11 0

physical page number page offset

physical address
Integrating VM and cache

- Conceptually, caches bring **physical** memory space closer:
  - Translation of VA to PA must be done before accessing caches
  - Translation *requires a memory access!*
  - *Use of TLB reduces to (still) 1 cycle*
- Called **physical cache**
Integrating VM and cache (2)

- Alternative solution:
  - Address cache with VA
  - Called **virtual caches**
    - Translation required for misses only

- Problems:
  - **Aliasing**:
    - 2 virtual addresses that point to the same physical page (e.g., shared pages).
    - Two cache blocks for one physical location
  - **Cache flushes** due to process switch:
    - Get false hits
    - Cost for flushing + “compulsory” misses from empty cache
  - **I/O**? (uses physical addresses)
Integrating VM and cache (3)

✧ Solution to aliases
  ✧ HW guarantees that every cache block has unique physical address
  ✧ SW guarantee
    • Lower n bits must have same address (called page coloring)

✧ Solution to cache flushes
  ✧ Add process identifier tag that identifies process as well as address within process
  ✧ Can’t get a hit if wrong process

✧ Solution to I/O
  ✧ Inverse map addresses…
Cache for 4 pages

Physical PageNo | Virtual PageNo
---|---
0101 | 0000
  | 1011

Same physical page maps to two different virtual pages in two different cache locations

Virtual PageNo | Physical PageNo
---|---
0000 | 0101
  | 1100

Same physical page maps to two different virtual pages *but* no alias in cache!
Integrating VM and cache (4)

- TLB and cache access can be done in parallel!
  - The LSB of the virtual address are not modified by the translation process
    - Can be used to start the cache access.
    - The page offset part of the VA used to index into the cache at the start of the cycle
  - TLB access proceeds as normal, using the Virtual Page portion of the virtual address
  - At the end of the cycle, the virtual page is translated into a physical frame number (assuming a TLB hit)

- Partially-virtually addressed caches
  - Virtual index, physical tag
Integrating VM and cache (5)

- Details of parallel accesses:

```
VA : Virtual Page   Page Offset

TLB Tag Store

TLB Data Store

Cache Tag Store

Cache Data Store

TLB Hit

TLB entry

Physical Frame Number

Cache Hit

PA : Physical Frame   Page Offset

Data
```
Example of real memory hierarchies
Alpha 21164 Hierarchy

- **Reg.s.**
- **L1 Data**
  - 1 cycle latency
  - 8KB, direct
  - Write-through
  - Dual Ported
  - 32B lines
- **L1 Instruction**
  - 8KB, direct
  - 32B lines
- **Processor Chip**
- **L2 Unified**
  - 8 cycle latency
  - 96KB
  - 3-way assoc.
  - Write-back
  - Write allocate
  - 32B/64B lines
- **L3 Unified**
  - 1M-64M
direct
  - Write-back
  - Write allocate
  - 32B or 64B lines
- **Main Memory**
  - Up to 1TB

---

Up to 1TB
Pentium memory hierarchy