CPE 631 Lecture 06: Cache Design

Aleksandar Milenkovic, milenka@ece.uah.edu
Electrical and Computer Engineering
University of Alabama in Huntsville

Outline
- Cache Performance
- How to Improve Cache Performance

Review: Caches
- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Capacity Misses: increase cache size
  - Conflict Misses: increase cache size and/or associativity
- Write Policy:
  - Write Through: needs a write buffer.
  - Write Back: control can be complex
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?

Review: The Cache Design Space
- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (i-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins
AMAT and Processor Performance

- Miss-oriented Approach to Memory Access
  - CPI_{EAMC} includes ALU and Memory instructions

\[
\text{CPU time} = \frac{IC \times (\text{CPI}_{EAMC} + \frac{\text{MemAccessInst}}{\text{Inst}} \times \text{MissRate} \times \text{MissPenalty})}{\text{Clockrate}}
\]

\[
\text{CPU time} = \frac{IC \times (\text{CPI}_{EAMC} + \frac{\text{MemMissesInst}}{\text{Inst}} \times \text{MissPenalty})}{\text{Clockrate}}
\]

Separating out Memory component entirely
- AMAT = Average Memory Access Time
- CPI_{ALUops} does not include memory instructions

\[
\text{CPU time} = \frac{IC \times (\frac{\text{ALUopsInst}}{\text{Inst}} + \frac{\text{MemAccessInst}}{\text{Inst}} \times \text{AMAT})}{\text{Clockrate}}
\]

Where Misses Come From?

- Classifying Misses: 3 Cs
  - 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 in even an Infinite Cache)
  - Capacity — If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. (Misses in Fully Associative Size X Cache)
  - Conflict — If block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses. (Misses in N-way Associative, Size X Cache)
  - Coherence — Misses caused by cache coherence.
3Cs Absolute Miss Rate (SPEC92)

- 8-way: conflict misses due to going from fully associative to 8-way assoc.
- 4-way: conflict misses due to going from 8-way to 4-way assoc.
- 2-way: conflict misses due to going from 4-way to 2-way assoc.
- 1-way: conflict misses due to going from 2-way to 1-way assoc. (direct mapped)

Cache Organization?

- Assume total cache size not changed
- What happens if:
  1) Change Block Size
  2) Change Cache Size
  3) Change Cache Internal Organization
  4) Change Associativity
  5) Change Compiler
- Which of 3Cs is obviously affected?

3Cs Relative Miss Rate

1st Miss Rate Reduction Technique: Larger Block Size

- Increased Conflict Misses

Block Size (bytes)
1st Miss Rate Reduction Technique:
Larger Block Size (cont’d)

- Example:
  - Memory system takes 40 clock cycles of overhead, and then delivers 16 bytes every 2 clock cycles
  - Miss rate vs. block size (see table); hit time is 1 cc
  - AMAT = Hit Time + Miss Rate x Miss Penalty

- Block size depends on both latency and bandwidth of lower level memory
- Low latency and bandwidth => decrease block size
- High latency and bandwidth => increase block size

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>1K</th>
<th>4K</th>
<th>16K</th>
<th>256K</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS</td>
<td>30</td>
<td>25</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>BS</td>
<td>35</td>
<td>30</td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>BS</td>
<td>40</td>
<td>35</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>BS</td>
<td>45</td>
<td>40</td>
<td>35</td>
<td>30</td>
</tr>
<tr>
<td>BS</td>
<td>50</td>
<td>45</td>
<td>40</td>
<td>35</td>
</tr>
<tr>
<td>BS</td>
<td>60</td>
<td>50</td>
<td>45</td>
<td>40</td>
</tr>
<tr>
<td>BS</td>
<td>65</td>
<td>55</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>BS</td>
<td>70</td>
<td>60</td>
<td>55</td>
<td>50</td>
</tr>
<tr>
<td>BS</td>
<td>80</td>
<td>65</td>
<td>60</td>
<td>55</td>
</tr>
<tr>
<td>BS</td>
<td>90</td>
<td>70</td>
<td>65</td>
<td>60</td>
</tr>
<tr>
<td>BS</td>
<td>100</td>
<td>75</td>
<td>70</td>
<td>65</td>
</tr>
</tbody>
</table>

2nd Miss Rate Reduction Technique:
Larger Caches

- Reduce Capacity misses
- Drawbacks: Higher cost, Longer hit time

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compulsory</td>
<td>0.02</td>
<td>0.04</td>
<td>0.06</td>
<td>0.08</td>
<td>0.1</td>
<td>0.12</td>
<td>0.14</td>
<td>0.16</td>
</tr>
</tbody>
</table>

3rd Miss Rate Reduction Technique:
Higher Associativity

- Miss rates improve with higher associativity
- Two rules of thumb
  - 8-way set-associative is almost as effective in reducing misses as fully-associative cache of the same size
  - 2:1 Cache Rule: Miss Rate DM cache size N = Miss Rate 2-way cache size N/2
- Beware: Execution time is only final measure!
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compulsory</td>
<td>0.02</td>
<td>0.04</td>
<td>0.06</td>
<td>0.08</td>
<td>0.1</td>
<td>0.12</td>
<td>0.14</td>
<td>0.16</td>
</tr>
</tbody>
</table>
3rd Miss Rate Reduction Technique: Higher Associativity (cont'd)

- **Example**
  - CCT_{2-way} = 1.10 \times CCT_{1-way}
  - CCT_{4-way} = 1.12 \times CCT_{1-way}
  - CCT_{8-way} = 1.14 \times CCT_{1-way}
  - Hit time = 1 cc, Miss penalty = 50 cc
  - Find AMAT using miss rates from Fig 5.9 (old textbook)

<table>
<thead>
<tr>
<th>CSize [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>1.44</td>
<td>1.42</td>
<td>1.45</td>
<td>1.50</td>
</tr>
<tr>
<td>2</td>
<td>1.79</td>
<td>1.77</td>
<td>1.80</td>
<td>1.95</td>
</tr>
<tr>
<td>4</td>
<td>3.50</td>
<td>3.00</td>
<td>2.87</td>
<td>2.59</td>
</tr>
<tr>
<td>8</td>
<td>5.44</td>
<td>5.25</td>
<td>5.13</td>
<td>5.04</td>
</tr>
<tr>
<td>64</td>
<td>3.19</td>
<td>3.57</td>
<td>3.95</td>
<td>4.60</td>
</tr>
<tr>
<td>128</td>
<td>1.50</td>
<td>1.45</td>
<td>1.42</td>
<td>1.44</td>
</tr>
</tbody>
</table>

4th Miss Rate Reduction Technique: Way Prediction, Pseudo-Associativity

- **Pseudo-Associative Cache**
  - Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)
  - Accesses proceed just as in the DM cache for a hit
  - On a miss, check the second entry
    - Simple way is to invert the MSB bit of the INDEX field to find the other block in the 'pseudo set'

<table>
<thead>
<tr>
<th>Hit Time</th>
<th>Pseudo Hit Time</th>
<th>Miss Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 cc</td>
<td>3 cc</td>
<td>50 cc</td>
</tr>
</tbody>
</table>

- **Way Prediction**
  - Extra bits are kept to predict the way or block within a set
  - Mux is set early to select the desired block
  - Only a single tag comparison is performed
  - What if miss?
  - Check the other blocks in the set

<table>
<thead>
<tr>
<th>CSize [KB]</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>Miss Rate ps.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.30</td>
<td>4.90</td>
<td>4.84</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
<td>1.45</td>
<td>1.356</td>
<td></td>
</tr>
</tbody>
</table>
### 5th Miss Rate Reduction Technique: Compiler Optimizations

- **Reduction comes from software (no Hw ch.)**
- McFarling [1989] reduced caches misses by 75% (8KB, DM, 4 byte blocks) in software
  - **Instructions**
    - Reorder procedures in memory so as to reduce conflict misses
    - Profiling to look at conflicts (using tools they developed)
- **Data**
  - Merging Arrays: Improve spatial locality by single array of compound elements vs. 2 arrays
  - 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

### Loop Interchange Example

#### /* 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; improved spatial locality.

Reduces misses if the arrays do not fit in the cache.

### Loop Interchange

- **Motivation:** some programs have nested loops that access data in nonsequential order
- **Solution:** Simply exchanging the nesting of the loops can make the code access the data in the order it is stored => reduce misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded

### Blocking

- **Motivation:** multiple arrays, some accessed by rows and some by columns
- **Solution:** instead of operating on entire rows and columns of an array, blocked algorithms operate on submatrices or blocks => maximize accesses to the data loaded into the cache before the data is replaced
Blocking Example

/* 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)
{z = z + y[i][k]*z[k][j];
x[i][j] = z;
};
};

Two Inner Loops:
- Read all N\times N elements of y[]
- Read N elements of 1 row of z[] repeatedly
- Write N elements of 1 row of x[]

Capacity Misses - a function of N & Cache Size:
- $2N^3 + N^2$ (assuming no conflict; otherwise ...)

Idea: compute on $B\times B$ submatrix that fits

阻locking Example (cont'd)

/* 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 $N^3/B + 2N^2$
- Conflict Misses Too?

Merging Arrays

Motivation: some programs reference multiple arrays in the same dimension with the same indices at the same time => these accesses can interfere with each other, leading to conflict misses

Solution: combine these independent matrices into a single compound array, so that a single cache block can contain the desired elements

Merging Arrays Example

/* 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];
Some programs have separate sections of code that access with the same loops, performing different computations on the common data.

Solution:

‘Fuse’ the code into a single loop =>
the data that are fetched into the cache can be used repeatedly before being swapped out => reducing misses via improved temporal locality.

---

### Loop Fusion Example

**Before**

```c
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**

```c
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; improve temporal locality.

---

### Summary of Compiler Optimizations to Reduce Cache Misses (by hand)

<table>
<thead>
<tr>
<th>Performance Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>compress</td>
</tr>
<tr>
<td>cholesky (nasa7)</td>
</tr>
<tr>
<td>spice (nasa7)</td>
</tr>
<tr>
<td>mxm (nasa7)</td>
</tr>
<tr>
<td>btrix (nasa7)</td>
</tr>
<tr>
<td>tomcatv</td>
</tr>
<tr>
<td>gmty (nasa7)</td>
</tr>
<tr>
<td>vpenta (nasa7)</td>
</tr>
<tr>
<td>merged arrays</td>
</tr>
<tr>
<td>loop array</td>
</tr>
<tr>
<td>loop fusion</td>
</tr>
<tr>
<td>blocking</td>
</tr>
</tbody>
</table>

---

### Summary: Miss Rate Reduction

3 Cs: Compulsory, Capacity, Conflict

- 1. Larger Cache => Reduce Capacity
- 2. Larger Block Size => Reduce Compulsory
- 3. Higher Associativity => Reduce Conflicts
- 4. Way Prediction & Pseudo-Associativity
- 5. Compiler Optimizations

**CPU time =**

\[
\text{IC} \times \left( \frac{\text{CPI}_{\text{Exec}}}{\text{Inst}} \times \frac{\text{MissRate}_{\text{MissPenalty}}}{\text{Clockrate}} \right)
\]
Reducing Miss Penalty

**Motivation**
- $AMAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$
- Technology trends $\Rightarrow$ relative cost of miss penalties increases over time

**Techniques that address miss penalties**
- 1. Multilevel Caches
- 2. Critical Word First and Early Restart
- 3. Giving Priority to Read Misses over Writes
- 4. Merging Write Buffer
- 5. Victim Caches

---

1st Miss Penalty Reduction Technique: Multilevel Caches

**Architect’s dilemma**
- Should I make the cache faster to keep pace with the speed of CPUs
- Should I make the cache larger to overcome the widening gap between CPU and main memory

**L2 Equations**
- $AMAT = Hit\ Time_{L2} + Miss\ Rate_{L2} \times Miss\ Penalty_{L2}$
- $Miss\ Penalty_{L2} = Hit\ Time_{L2} + Miss\ Rate_{L2} \times Miss\ Penalty_{L2}$
- $AMAT = Hit\ Time_{L1} + Miss\ Rate_{L1} \times (Hit\ Time_{L2} + Miss\ Rate_{L2} + Miss\ Penalty_{L2})$

**Definitions:**
- Local miss rate — misses in this cache divided by the total number of memory accesses to this cache ($Miss\ Rate_{L1}$)
- Global miss rate — misses in this cache divided by the total number of memory accesses generated by the CPU ($Miss\ Rate_{L2}$)
- Global Miss Rate is what matters

---

Reducing Misses: Which apply to L2 Cache?

**Reducing Miss Rate**
- 1. Reduce Capacity Misses via Larger Cache
- 2. Reduce Compulsory Misses via Larger Block Size
- 3. Reduce Conflict Misses via Higher Associativity
- 4. Reduce Conflict Misses via Way Prediction & Pseudo-Associativity
- 5. Reduce Conflict/Capac. Misses via Compiler Optimizations
### Relative CPU Time

<table>
<thead>
<tr>
<th>Block Size</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.36</td>
<td>1.28</td>
<td>1.27</td>
<td>1.34</td>
<td>1.54</td>
<td>1.95</td>
</tr>
</tbody>
</table>

### L2 cache block size & A.M.A.T.

- 32KB L1, 8 byte path to memory

### 2nd Miss Penalty Reduction Technique: Early Restart and Critical Word First

- Don’t wait for full block to be loaded before restarting CPU
  - 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—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first
- Generally useful only in large blocks
- Problem of spatial locality: tend to want next sequential word, so not clear if benefit by early restart and CWF

### Multilevel Inclusion: Yes or No?

- Inclusion property:
  - L1 data are always present in L2
    - Good for I/O & caches consistency
    - (L1 is usually WT, so valid data are in L2)
- Drawback: What if measurements suggest smaller cache blocks for smaller L1 caches and larger blocks for larger L2 caches?
  - E.g., Pentium4: 64B L1 blocks, 128B L2 blocks
  - Add complexity: when replace a block in L2 should discard 2 blocks in the L1 cache => increase L1 miss rate
- What if the budget for a L2 cache is slightly bigger than the L1 cache => L2 keeps redundant copy of L1
  - Multilevel Exclusion: L1 data is never found in a L2 cache
  - E.g., AMD Athlon uses this:
    - 64KB L1$ + 64KB L1D$ vs. 256KB L2U$

### 3rd Miss Penalty Reduction Technique: Giving Read Misses Priority over Writes

- Tag
- Write buffer
- Delayed Write Buffer
- Lower level memory
- CPU Address
- Data in
- Address
- Data
- Write buffer
- CPU
3rd Miss Penalty Reduction Technique: Read Priority over Write on Miss (2)

- Write-through with write buffers offer RAW conflicts with main memory reads on cache misses
  
  Example: DM, WT, 512 & 1024 map to the same block:
  
  DM 512(R0), R3 ; cache index 0
  LW R1, 1024(R0) ; cache index 0
  LW R2, 512(R0) ; cache index 0

  - If simply wait for write buffer to empty, might increase read miss penalty (old MIPS 1000 by 50%)
  - Check write buffer contents before read; if no conflicts, let the memory access continue

- Write-back also want buffer to hold misplaced blocks
  
  - Read miss replacing dirty block
  
  - Normal: Write dirty block to memory, and then do the read
  
  - Instead copy the dirty block to a write buffer, then do the read, and then do the write
  
  - CPU stall less since restarts as soon as do read

4th Miss Penalty Reduction Technique: Merging Write Buffer

- Write Through caches relay on write-buffers
  
  - on write, data and full address are written into the buffer; write is finished from the CPU’s perspective
  
  - Problem: WB full stalls

- Write merging
  
  - multword writes are faster than a single word writes => reduce write-buffer stalls

  - Is this applicable to I/O addresses?

5th Miss Penalty Reduction Technique: Victim Caches

- How to combine fast hit time of direct mapped yet still avoid conflict misses?

  - Idea: Add buffer to place data discarded from cache in the case it is needed again

  - Jouppi (1990):
    
    4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache

  - Used in Alpha, HP machines, AMD Athlon (8 entries)

Summary of Miss Penalty Reducing Tec.

- 1. Multilevel Caches
- 2. Critical Word First and Early Restart
- 3. Giving Priority to Read Misses over Writes
- 4. Merging Write Buffer
- 5. Victim Caches
Reducing Cache Miss Penalty or Miss Rate via Parallelism

- Idea: overlap the execution of instructions with activity in memory hierarchy
- Miss Rate/Penalty reduction techniques
  - 1. Nonblocking caches
    - reduce stalls on cache misses
      in CPUs with out-of-order completion
  - 2. Hardware prefetching of instructions and data
    - reduce miss penalty
  - 3. Compiler controlled prefetching

Reduce Misses/Penalty: Non-blocking Caches to reduce stalls on misses

- Non-blocking cache or lockup-free cache allow data cache to continue to supply cache hits during a miss
  - requires F/E bits on registers or out-of-order execution
  - requires multi-bank memories
- “hit under miss” reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- “hit under multiple miss” or “miss under miss” may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller
  - Requires multiple memory banks (otherwise cannot support)
  - Pentium Pro allows 4 outstanding memory misses

Reducing Misses/Penalty by Hardware Prefetching of Instructions & Data

- E.g., Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
    - Extra block placed in “stream buffer”
    - On miss check stream buffer
  - Works with data blocks too:
    - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
    - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches
- Prefetching relies on having extra memory bandwidth that can be used without penalty
Reducing Misses/Penalty by Software Prefetching Data

- Data Prefetch
  - Load data into register (HP PA-RISC loads)
  - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
  - Special prefetching instructions cannot cause faults; a form of speculative execution

- Prefetching comes in two flavors:
  - Binding prefetch: Requests load directly into register.
    - Must be correct address and register!
  - Non-Binding prefetch: Load into cache.
    - Can be incorrect. Faults?

- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth

1st Hit Time Reduction Technique: Small and Simple Caches

- Smaller hardware is faster =>
  - Small cache helps the hit time

- Keep the cache small enough to fit on the same chip as the processor (avoid the time penalty of going off-chip)

- Keep the cache simple
  - Use Direct Mapped cache: it overlaps the tag check with the transmission of data

Review: Improving Cache Performance

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

AMAT = HitTime + MissRate · MissPenalty

2nd Hit Time Reduction Technique: Avoiding Address Translation

- Conventional Organization
- Virtually Addressed Cache

Overlap $ access with VA translation: requires $ index to remain invariant across translation
2nd Hit Time Reduction Technique: Avoiding Address Translation (cont’d)

- Send virtual address to cache? Called Virtually Addressed Cache or just Virtual Cache vs. Physical Cache
  - Every time process is switched logically must flush the cache; otherwise get false hits
  - Cost is time to flush + “compulsory” misses from empty cache
- Dealing with aliases (sometimes called synonyms): Two different virtual addresses map to same physical address => multiple copies of the same data in a virtual cache
  - I/O typically uses physical addresses; if I/O must interact with cache, mapping to virtual addresses is needed
- Solution to aliases
  - HW solutions guarantee every cache block a unique physical address
- Solution to cache flush
  - Add process identifier tag that identifies process as well as address within process: can’t get a hit if wrong process

Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>KR</th>
<th>SP</th>
<th>HIT</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smaller Block Size</td>
<td>+</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>+</td>
<td>-</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Part Caches</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Node-Addressable Caches</td>
<td>+</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Prefetching of Instr/Data</td>
<td>+</td>
<td>+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Compiler-controlled Prefetching</td>
<td>+</td>
<td>+</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td>+</td>
<td>+</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Early Start of pipeline</td>
<td>+</td>
<td>-</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Early Restart of pipeline before 1st</td>
<td>+</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>HW Blocking Caches</td>
<td>+</td>
<td>-</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Second Level Caches</td>
<td>+</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Core Memory Systems</td>
<td>+</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Real &amp; Simple Caches</td>
<td>-</td>
<td>+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Stacking Address Translation</td>
<td>+</td>
<td>+</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

2/2/2004 UAH-CPE631 53