CPE 631 Lecture 07: Cache Design & Main Memory

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:
  - 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\text{EAC} includes ALU and Memory instructions
  \[ \text{CPU time} = \frac{IC \times (\text{CPI}_{\text{E}}, \text{Inst}) + \text{MemAccess}, \text{Inst} \times \text{MissRate}, \text{MissPenalty}}{\text{Clockrate}} \]
  \[ \text{CPU time} = \frac{IC \times (\text{CPI}_{\text{E}}, \text{Inst}) \times \text{MemMisses}, \text{Inst} \times \text{MissPenalty}}{\text{Clockrate}} \]

AMAT and Processor Performance (cont’d)

- Separating out Memory component entirely
  - AMAT = Average Memory Access Time
  - CPI\text\{ALU\text{Ops}\} does not include memory instructions
  \[ \text{CPU time} = \frac{\text{ALUOps}, \text{Inst} \times \text{CPI}_{\text{ALU}, \text{Inst}} + \text{MemAccess}, \text{Inst} \times \text{AMAT}}{\text{Clockrate}} \]

How to Improve Cache Performance?

- AMAT = HitTime + MissRate \times MissPenalty
- Cache optimizations
  - 1. Reduce the miss rate
  - 2. Reduce the miss penalty
  - 3. Reduce the time to hit in the cache

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)
  - More recent, 4th “C”:
    - Coherence — Misses caused by cache coherence.

Aleksandar Milenkovich
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)

3Cs Relative Miss Rate

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?

1st Miss Rate Reduction Technique: Larger Block Size

- Larger Block Size
  - Reduced compulsory misses
  - Increased conflict misses
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

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

2nd Miss Rate Reduction Technique: Larger Caches

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

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%
Miss Rate Reduction Technique: Higher Associativity (cont’d)

- Example
  - \( CCT_{2\text{-way}} = 1.10 \times CCT_{1\text{-way}} \)
  - \( CCT_{4\text{-way}} = 1.12 \times CCT_{1\text{-way}} \)
  - \( CCT_{8\text{-way}} = 1.14 \times CCT_{1\text{-way}} \)
  - 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>1.44</td>
</tr>
<tr>
<td>2</td>
<td>1.59</td>
<td>1.57</td>
<td>1.60</td>
<td>1.59</td>
</tr>
<tr>
<td>4</td>
<td>1.70</td>
<td>1.60</td>
<td>1.67</td>
<td>1.59</td>
</tr>
<tr>
<td>16</td>
<td>1.50</td>
<td>1.45</td>
<td>1.47</td>
<td>1.44</td>
</tr>
</tbody>
</table>

Miss Rate Reduction Technique: Way Prediction, “Pseudo-Associativity”

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
  - **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
    - Used in Alpha 21264 (1 bit per block in IC$)  
      - 1 cc if predictor is correct, 3 cc if not
    - Effectiveness: prediction accuracy is 85%
    - Used in MIPS 4300 embedded proc. to lower power

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>CSize [KB]</th>
<th>1-way</th>
<th>2-way</th>
<th>Pseudo-Hit-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.30</td>
<td>3.00</td>
<td>2.87</td>
</tr>
<tr>
<td>16</td>
<td>3.45</td>
<td>3.35</td>
<td>3.19</td>
</tr>
<tr>
<td>64</td>
<td>3.70</td>
<td>3.60</td>
<td>3.77</td>
</tr>
<tr>
<td>128</td>
<td>3.50</td>
<td>3.45</td>
<td>3.67</td>
</tr>
</tbody>
</table>

Pseudo-Assocativity

- Compare 1-way, 2-way, and pseudo associative organizations for 2KB and 128KB caches
- Hit time = 1cc, Pseudo hit time = 2cc
- Parameters are the same as in the previous Exmp.
  - AMAT\(_{ps}\) = Hit Time\(_{ps}\) + Miss Rate\(_{ps}\) x Miss Penalty\(_{ps}\)
  - Miss Rate\(_{ps}\) = Miss Rate\(_{2\text{-way}}\)
  - Hit time\(_{ps}\) = Hit time\(_{2\text{-way}}\) + Alternate hit rate\(_{2\text{-way}}\) \times 2
  - Alternate hit rate\(_{2\text{-way}}\) = Hit rate\(_{2\text{-way}}\) - Hit rate\(_{1\text{-way}}\)

<table>
<thead>
<tr>
<th>CSize [KB]</th>
<th>1-way</th>
<th>2-way</th>
<th>Pseudo</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5.30</td>
<td>4.90</td>
<td>4.944</td>
</tr>
<tr>
<td>128</td>
<td>1.50</td>
<td>1.45</td>
<td>1.356</td>
</tr>
</tbody>
</table>
### 5th Miss Rate Reduction Technique: Compiler Optimizations

- **Reduction comes from software (no HW ch.).**
- McFarling [1989] reduced cache 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

- **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.

#### Loop Interchange Example

```c
/* 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.

### Blocking

- **Motivation:** Multiple arrays, some accessed by rows and some by columns.
- **Storing the arrays row by row (row major order) or column by column (column major order) does not help:** Both rows and columns are used in every iteration of the loop (Loop Interchange cannot help).
- **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**
```c
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:** a function of N & Cache Size:
- $2N^3 + N^2$ (assuming no conflict; otherwise ...)

**Idea:** compute on BxB submatrix that fits

**After**
```c
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

**Example**
```c
/* 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

```c
/* 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; 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>1</td>
</tr>
<tr>
<td>compress</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
Reducing Miss Penalty

- **Motivation**
  - AMAT = Hit Time + Miss Rate x Miss Penalty
  - Technology trends => 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\_L1 + Miss Rate\_L1 x Miss Penalty\_L1
  - Miss Penalty\_L1 = Hit Time\_L2 + Miss Rate\_L2 x Miss Penalty\_L2
  - AMAT = Hit Time\_L1 x (Miss Rate\_L1 + Miss Rate\_L2) + Miss Penalty\_L1

- **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\_L1 x 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
L2 cache block size & A.M.A.T.

- 32KB L1, 8 byte path to memory

<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>Relative CPU Time</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>

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$

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

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

- Tag
- Data
- Delayed Write Buffer
- CPU Address
- Lower level memory

- 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
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

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
  - multiword writes are faster than a single word writes => reduce write-buffer stalls
- Is this applicable to I/O addresses?

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 as there can be multiple outstanding memory accesses
  - Requires multiple memory banks (otherwise cannot support)
  - Pentium Pro allows 4 outstanding memory misses

Value of Hit Under Miss for SPEC

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

Review: Improving Cache Performance

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

\[ AMAT = \text{HitTime} + \text{MissRate} \cdot \text{MissPenalty} \]
3rd Hit Time Reduction Technique: Pipelined Cache Access

- Pipeline cache accesses
  - Pentium: 1 cc cache hit
  - Pentium Pro – Pentium III – 2 cc cache hit
  - Pentium4 – 4 cc cache hit
- Increase in the number of pipeline stages => Larger penalty for branch mispredictions

4th Hit Time Reduction Technique: Trace Cache

- Pentium4 (NetBurst Architecture) – L1 cache has been replaced with TraceCache
- Instead of static cache blocks, trace cache finds the sequence of instructions including taken branches
- branch prediction is folded into the cache
- Downside: one instruction can be stored multiple times (in different traces)

NetBurst’s Execution Trace Cache

- Keeps micro-operations (decoded IA32 instructions)
  - Capacity – 12K µops
  - Hit rate is similar to an 8KB to 16KB I$
  - Delivers up to 3 µops per clock cycle
- Dynamically build traces – program-ordered sequences of µops
  - Packs µops into groups of 6 per trace line
    - one trace line can include branch and its target even if they are thousands of bytes apart in the program
  - One trace can include many trace lines! How many?
- Has its own branch predictor
# Cache Optimization Summary

<table>
<thead>
<tr>
<th>Metric</th>
<th>UR</th>
<th>AP</th>
<th>HT</th>
<th>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>+1</td>
<td>+1</td>
</tr>
<tr>
<td>Level Caches</td>
<td>+</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Anti-Associative Caches</td>
<td>+</td>
<td>+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Offloading Displacement of Data</td>
<td>+</td>
<td>+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Higher Control of Preempt</td>
<td>+</td>
<td>+</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Faster Preempt的Meeses</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Early Restart &amp; Critical Word 1st</td>
<td>+</td>
<td>+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Prior Block Cache</td>
<td>+</td>
<td>+</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Second Level Cache</td>
<td>+</td>
<td>+</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Better memory system</td>
<td>+</td>
<td>+</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Hard &amp; Simple Caches</td>
<td>+</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Faster Address Translation</td>
<td>+</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Testing Caches</td>
<td>+</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

## Main Memory

### Main Memory Background
- Next level down in the hierarchy
  - satisfies the demands of caches + serves as the I/O interface
- Performance of Main Memory:
  - **Latency**: Cache Miss Penalty
    - Access Time: time between when a read is requested and when the desired word arrives
    - Cycle Time: minimum time between requests to memory
    - Bandwidth: (the number of bytes read or written per unit time):
      - I/O & Large Block Miss Penalty (5.2)
- Main Memory is **DRAM**: Dynamic Random Access Memory
  - Dynamic since needs to be refreshed periodically (8 ms, 1% time)
  - Addresses divided into 2 halves (Memory as a 2D matrix):
    - RAS or Row Access Stride + CAS or Column Access Stride
- Cache uses **SRAM**: Static Random Access Memory
  - No refresh (6 transistors/bit vs. 1 transistor)
Memory Background:
Dynamic RAM

- SRAM cells exhibit high speed/poor density
- DRAM: simple transistor/capacitor pairs in high density form

Memory Background:
Dynamic RAM

Techniques for Improving Performance

- 1. Wider Main Memory
- 2. Simple Interleaved Memory
- 3. Independent Memory Banks

Memory Organizations

- Simple: CPU, Cache, Bus, Memory same width (32 or 64 bits)
- Wide: CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits; UltraSPARC 512 bits)
- Interleaved: CPU, Cache, Bus 1 word; Memory N Modules (4 Modules); example is word interleaved

1st Technique for Higher Bandwidth:
Wider Main Memory (cont’d)

- Timing model (word size is 8 bytes = 64 bits)
  - 4cc to send address, 56cc for access time per word, 4cc to send data
  - Cache Block is 4 words
- Simple M.P.: \(4 \times (4+56+4) = 256 \text{ cc} (1/8 \text{ B/cc})\)
- Wide M.P.(2W): \(2 \times (4+56+4) = 128 \text{ cc} (1/4 \text{ B/cc})\)
- Wide M.P.(4W): \(4+56+4 = 64 \text{ cc} (1/2 \text{ B/cc})\)
2nd Technique for Higher Bandwidth: Simple Interleaved Memory

- Take advantage of potential parallelism of having many chips in a memory system
  - Memory chips are organized in banks allowing multi-word read or writes at a time
- Interleaved M.P. = 4 + 56 + 4x4 = 76 cc (0.4B/cc)

<table>
<thead>
<tr>
<th>Rank 0</th>
<th>Rank 1</th>
<th>Rank 2</th>
<th>Rank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

How many banks?
- number banks ≤ number clocks to access word in bank
- For sequential accesses, otherwise will return to original bank before it has next word ready
- Consider the following example: 10cc to read a word from a bank, 8 banks

Problem #1: Chip size increase
- 512MB DRAM using 4Mx4bits: 256 chips => easy to organize in 16 banks with 16 chips
- 512MB DRAM using 64Mx4bits: 16 chips => 1 bank?

Problem #2: Difficulty in main memory expansion

3rd Technique for Higher Bandwidth: Independent Memory Banks

- Memory banks for independent accesses vs. faster sequential accesses
  - Multiprocessor
  - I/O
  - CPU with Hit under n Misses, Non-blocking Cache
- Superbank: all memory active on one block transfer (or Bank)
- Bank: portion within a superbank that is word interleaved (or Subbank)

Avoiding Bank Conflicts

- Lots of banks
  - Even with 128 banks, for (i = 0; i < 512; i = i + 1) since 512 is multiple of 128, conflict on word accesses
  - SW: loop interchange or declaring array not power of 2 ("array padding")
- HW: Prime number of banks
  - bank number = address mod number of banks
  - address within bank = address / number of words in bank
  - modulo & divide per memory access with prime no. banks?
  - address within bank = address mod number words in bank
  - bank number? easy if 2N words per bank

Aleksandar Milenkovich
Fast Bank Number

- Chinese Remainder Theorem: As long as two sets of integers $a_i$ and $b_i$ follow these rules
  
  \[ b_i = a_i \mod a_j, \quad 1 \leq i < j \leq n \]
  
  - $a_i$ and $a_j$ are co-prime if $i \neq j$,
  - then the integer $x$ has only one solution (unambiguous mapping):
    - bank number $= b_0$,
    - number of banks $= a_0$ (= 3 in example)
    - address within bank $= b_1$,
    - number of words in bank $= a_1$ (= 8 in ex.)
    - $N$ word address 0 to $N-1$, prime no. banks, words power of 2

<table>
<thead>
<tr>
<th>Rank Number</th>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>15</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
<td>0</td>
<td>15</td>
<td>7</td>
</tr>
</tbody>
</table>

Seq. Interleaved Modulo Interleaved

DRAM logical organization (64 Mbit)

Square root of bits per RAS/CAS

4 Key DRAM Timing Parameters

- $t_{RAC}$: minimum time from RAS line falling to the valid data output
  - Quoted as the speed of a DRAM when buy
  - A typical 4Mb DRAM $t_{RAC} = 60$ ns
  - Speed of DRAM since on purchase sheet?
- $t_{CAC}$: minimum time from the start of one row access to the start of the next
  - $t_{CAC} = 110$ ns for a 4Mb DRAM with $t_{RAC}$ of 60 ns
- $t_{PC}$: minimum time from CAS line falling to valid data output
  - 15 ns for a 4Mb DRAM with a $t_{CAC}$ of 60 ns
  - $t_{PC}$: minimum time from the start of one column access to the start of the next
  - 35 ns for a 4Mb DRAM with a $t_{CAC}$ of 60 ns

DRAM Read Timing

- Every DRAM access begins at:
  - The assertion of the RAS_L
  - 2 ways to read: early or late v. CAS

2/4/2004

Aleksandar Milenkovich
A 60 ns ($t_{DAC}$) DRAM can
- perform a row access only every 110 ns ($t_{RC}$)
- perform column access ($t_{CAC}$) in 15 ns, but time between column accesses is at least 35 ns ($t_{PC}$).
  - In practice, external address delays and turning around buses make it 40 to 50 ns.
These times do not include the time to drive the addresses off the microprocessor nor the memory controller overhead!

Fast Page Mode
- allow repeated access to the row buffer without another row access.
Improving Memory Performance via a New DRAM Interface: RAMBUS

RDRAM Memory System

RDRAM Bank