Chapter 5: Exploiting the Memory Hierarchy
Lecture 4

Lecturer: Rami Melhem
**Cache Controller FSM (sec. 5.9)**

- **Idle (cache ready)**: Cache hit / return data to CPU
- **Compare Tags & Valid bit**: Request from CPU
- **Request block from memory**: Memory ready
- **Write back old block**: Memory busy

**Software optimization 1: loop interchange (sec. 5.4)**

Matrix A is stored Row-wise (row major)
Fully associative cache, block size = 4 words
Cache size < 4n words

- for (j = 0; j < n; j = j+1)
- for (i = 0; i < n; i = i+1)
  
  \[ C = A[i][j]; \]

**Take advantage of spatial locality**

Row-wise memory access
- for (i = 0; i < n; i = i+1)
- for (j = 0; j < n; j = j+1)
  
  \[ C = A[i][j]; \]

100% Miss rate
25% Miss rate
Software optimization 2: blocking (partitioning)

Matrix multiplication

\[
C_{i,j} = \sum_{k} A_{i,k} \times B_{k,j}
\]

for \((i = 0; i < n; i++)\)
  for \((j = 0; j < n; j++)\)
    \(r = 0;\)
    for \((k = 0; k < n; k++)\)
      \(r = r + A_{i,k} \times B_{k,j};\)
    \(C_{i,j} = r;\)

Data used when \(i = 0, j = 0, \ldots, n-1\)

Assume:
A fully associative cache
Block size = 1 word
Cache size < \(n^2\)

- One row of \(A\) will fit in the cache and be repeatedly used (perfect reuse)
- \(B\) will not fit in cache and hence a column of \(B\) will be evicted before reuse
- Every element of \(B\) will be used only once when brought to the cache

Software optimization through blocking (partitioning)

Partition the matrices into submatrices of size \(p \times p\)

\[
C_{i,j} = \sum_{k=0}^{p-1} A_{i,k+p} \times B_{k+p,j}
\]

for \((si = 0; si < n; si += p)\)
  for \((sj = 0; sj < n; sj += p)\)
    for \((sk = 0; sk < n; sk += p)\)
      for \((i=si; i < si+p; i++)\)
        for \((j=sj; j < sj+p; j++)\)
          \(r = 0;\)
          for \((k=sk; k < sk+p; k++)\)
            \(r = r + A_{i,k} \times B_{k,j};\)
        \(C_{i,j} = C_{i,j} + r;\)

If cache size > \(p^2 \times n + p^2\), then
- \(A\) will be perfectly reused
- Each element of \(B\) will be reused “\(p\)” times (reduce miss rate)
HW optimization: Interleaving for faster block access

- Accessing a block of K words should be faster than accessing K words at different times. Otherwise, fetching an entire block does not pay off – actually may hurt.
- EX: 1 cycle to send a word or an address and 15 cycles to fetch a word in memory (only 7 cycles if “open row” policy and the row is already open in the row buffer)

<table>
<thead>
<tr>
<th>One-word wide memory</th>
<th>4-word wide organization (wide bus and memory ports)</th>
<th>Interleaved memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to access 4 words</td>
<td>Time to access a block (4 words)</td>
<td>Time to access a block (4 words)</td>
</tr>
<tr>
<td>4*[(1+15)+1]=68 cycles</td>
<td>(1+15+1)+3(1+7+1)=44 cycles</td>
<td>(1+15+1)+3=20 cycles</td>
</tr>
<tr>
<td>Rows will not be open if accesses are not strictly consecutive</td>
<td>The row will be open when accessing the last three words</td>
<td></td>
</tr>
</tbody>
</table>

- T = 1 send request to bank 0
- T = 2 send request to bank 1
- T = 3 send request to bank 2
- T = 4 send request to bank 3
- T = 16 data ready at Bank 0 and received at T = 17
- T = 17 data ready at Bank 0 and received at T = 18
- T = 18 data ready at Bank 0 and received at T = 19
- T = 19 data ready at Bank 0 and received at T = 20

Time to access memory = (1 + 15 + 1) + 3 = 20 cycles
**HW optimization: using write buffers to reduce miss penalty**

- Writing back an evicted block before reading a block increases the miss penalty
- Can read the requested block before writing back the old if a write buffer is used
- Priority is given to reading blocks in order to reduce miss penalty
- Blocks in the buffer are written back whenever there are no read requests
- Consecutive read requests will result in pending write backs in the write buffer

- For correctness, before sending any read request to memory, we have to check the write buffer
- If block is still in the write buffer, do not send the request to memory.

**Dependable memory hierarchy (sec. 5.5)**

- Fault: failure of a component
- Error: manifestation of a fault
- Faults may or may not lead to system failure

- **Reliability measure**: mean time to failure (MTTF)
- **Repair efficiency**: mean time to repair (MTTR)
- Mean time between failures
  \[
  \text{MTBF} = \text{MTTF} + \text{MTTR}
  \]
- Availability = MTTF / MTBF
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair
Error detection Codes (even parity codes)

- Hamming distance: Number of bits that are different between two bit patterns
  - Example: distance between 1001 and 1010 is 2
- Codes with min. distance = 2 provides single bit error detection.
  - Example: even parity code
    - Eight of the sixteen 4-bit code words are invalid
    - Any single bit flip in a valid code will produce an invalid code
      - Hence, single error detection --- but cannot correct the error
      - Note that two errors will go undetected

Error correcting Codes

Minimum distance = 3 provides single error correction (SEC)

Single fault (can be corrected)

Double faults (corrected to wrong code)
Error detection and correction

Minimum distance = 4 provides single error correction (SEC) and double error detection (DEC)

00 → 000000
01 → 001111
10 → 111001
11 → 110110

2-bit → 6-bit encoding

Single fault (can be corrected)

Double faults (will be detected)

Three faults (corrected to wrong code)

Hamming (Single error correcting) code

To calculate code a 12-bits code word from an 8-bits data word:
- Number the bits of the code words from 1 to 12
- All bit positions that are a power of 2 are parity bits, the others are data bits
- Each parity bit is set so that a certain group of data bits have even parity.

Encoded data bits

<table>
<thead>
<tr>
<th>Parity bit coverage</th>
<th>p1</th>
<th>p2</th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d5</th>
<th>d6</th>
<th>d7</th>
<th>d8</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p8</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:
Data bits 10100011 are encoded as 01100100011

Note: the minimum distance of the Hamming code = 3
Hamming (Single error correcting) code

- If an error occurs in any of the 12 bits
  
  **EXAMPLE:** 011001000011 becomes 011001000111

- Check the parity groups and compute the syndrome bits, \( s_1, s_2, s_3, s_4 \)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_2 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_3 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_4 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( s_1 = 0 \) (parity is correct)
- \( s_2 = 1 \) (parity is not correct)
- \( s_3 = 0 \) (parity is correct)
- \( s_4 = 1 \) (parity is not correct)

- If all syndrome bits are zeroes, then there is no error
- Otherwise, the syndrome bits indicate the position of the bit in error
- In our example, \( s_4, s_3, s_2, s_1 = 1010 \) \( \rightarrow \) bit ten is the wrong bit

Not magic!!
There is a theory behind that

---

Hamming SEC/DED Code

- Hamming code cannot detect two errors (distance < 4)
- Add an additional parity bit for the whole word (\( p_n \))
- Make Hamming distance = 4
- Decoding:
  - No error in \( p_n \) and syndrome = 0 \( \rightarrow \) no error
  - Error in \( p_n \) and syndrome > 0 \( \rightarrow \) single correctable error
  - No error in \( p_n \) and syndrome > 0 \( \rightarrow \) double errors (uncorrectable)
  - Error in \( p_n \) and syndrome = 0 \( \rightarrow \) error in the SEC parity bit

- Note: ECC DRAM uses SEC/DED with 8 bits (7 for syndrome and 1 for \( p_n \)) protecting each 64 bits