LECTURE 14
Caching and Cache-Efficient Algorithms

Charles E. Leiserson
October 29, 2013
CACHE HARDWARE
Multicore Cache Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Size</th>
<th>Assoc.</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>48 GB</td>
<td></td>
<td>200+</td>
</tr>
<tr>
<td>LLC</td>
<td>12 MB</td>
<td>16</td>
<td>40</td>
</tr>
<tr>
<td>L2</td>
<td>256 KB</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>L1–d</td>
<td>32 KB</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>L1–i</td>
<td>32 KB</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

64 B cache blocks

© 2013 Charles E. Leiserson and I–Ting Angelina Lee
Fully Associative Cache

Cache size $M = 32$
Line (block) size $B = 4$

A cache block can reside anywhere in the cache.

To find a block in the cache, the entire cache must be searched. When the cache becomes full, a block must be evicted to make room for a new block. The cache replacement policy determines which block to evict.
A cache block’s set determines its location in the cache.

To find a block in the cache, only a single location in the cache need be checked.

<table>
<thead>
<tr>
<th>address</th>
<th>tag_h</th>
<th>set</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w - \log M$</td>
<td>$\log(M/B)$</td>
<td>$\log B$</td>
<td></td>
</tr>
</tbody>
</table>
Set-Associative Cache

Cache size $M = 32$
Line (block) size $B = 4$
k = 2-way associativity

A cache block’s set determines $k$ possible locations in the cache.

To find a block in the cache, only the $k$ locations of its set must be checked.

<table>
<thead>
<tr>
<th>Address</th>
<th>tag_h</th>
<th>set</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w - \lg(M/k)$</td>
<td>$\lg(M/kB)$</td>
<td>$\lg B$</td>
<td></td>
</tr>
</tbody>
</table>
Taxonomy of Cache Misses

Cold miss
- The first time the cache block is accessed.

Capacity miss
- The previous cached copy would have been evicted even with a fully associative cache.

Conflict miss
- Too many blocks from the same set in the cache. The block would not have been evicted with a fully associative cache.

Sharing miss
- Another processor acquired exclusive access to the cache block.
  - True-sharing miss: The two processors are accessing the same data on the cache line.
  - False-sharing miss: The two processors are accessing different data that happen to reside on the same cache line.
Conflict Misses for Matrices

Conflict misses can be problematic for caches with limited associativity.

Assume:
- Word width $w = 64$
- Cache size $M = 32K$
- Line (block) size $B = 64$
- $k=4$-way associativity

Analysis
Look at a column of submatrix $A$. The addresses of the elements are $x, x+2^{15}, x+2\cdot2^{15}, \ldots, x+31\cdot2^{15}$. They all fall into the same set!

Solutions
Copy $A$ into a temporary $32 \times 32$ matrix, or pad rows.
IDEAL–CACHE MODEL
**Parameters**

- Two-level hierarchy.
- Cache size of $M$ bytes.
- Cache-line length of $B$ bytes.
- Fully associative.
- Optimal, omniscient replacement.

**Performance Measures**

- **work** $W$ (ordinary running time)
- **cache misses** $Q$
“LRU” Lemma [ST85]. Suppose that an algorithm incurs $Q$ cache misses on an ideal cache of size $M$. Then on a fully associative cache of size $2M$ that uses the least–recently used (LRU) replacement policy, it incurs at most $2Q$ cache misses. □

Implication
For asymptotic analyses, one can assume optimal or LRU replacement, as convenient.

Software Engineering
- Design a theoretically good algorithm.
- Engineer for detailed performance.
  - Real caches are not fully associative.
  - Loads and stores have different costs with respect to bandwidth and latency.
Cache-Miss Lemma

**Lemma.** Suppose that a program reads a set of \( r \) data segments, where the \( i \)th segment consists of \( s_i \) bytes, and suppose that

\[
\sum_{i=1}^{r} s_i = N < \mathcal{M}/3 \quad \text{and} \quad N/r \geq B.
\]

Then all the segments fit into cache, and the number of misses to read them all is at most \( 3N/B \).

**Proof.** A single segment \( s_i \) incurs at most \( s_i/B + 2 \) misses, and hence we have

\[
\sum_{i=1}^{r} (s_i/B + 2) = N/B + 2r
\]
\[
= N/B + (2rB)/B
\]
\[
\leq N/B + 2N/B
\]
\[
\leq 3N/B
\]
\[
\leq \mathcal{M}/B 
\]

\( \blacksquare \)
Tall Caches

Tall-cache assumption
\(B^2 < c M\) for some sufficiently small constant \(c \leq 1\).

Example: Intel Core i7 (Nehalem)
- Cache-line length = 64 bytes.
- L1-cache size = 32 Kbytes.
Submatrix Caching Lemma

**Theorem.** Suppose that an $n \times n$ submatrix is read into a tall cache satisfying $B^2 \leq cM$, where $c \leq 1$ is constant, and suppose that $cM \leq n^2 \leq M/3$. Then the submatrix fits into cache, and the number of misses to read all elements is at most $3n^2/B$.

**Proof.** Since we have $n = r = s_i$, $B \leq n = N/r$, and $N \leq M/3$, the Cache-Miss Lemma applies. ■
CACHE ANALYSIS OF MATRIX MULTIPLICATION
Multiply Square Matrices

void Mult(double *C, double *A, double *B, int n) {
    for (int i=0; i < n; i++)
        for (int j=0; j < n; j++)
            for (int k=0; k < n; k++)
                C[i*n+j] += A[i*n+k] * B[k*n+j];
}

Analysis of work
W(n) = Θ(n³).
Analysis of Cache Misses

```c
void Mult(double *C, double *A, double *B, int n) {
    for (int i=0; i < n; i++)
        for (int j=0; j < n; j++)
            for (int k=0; k < n; k++)
                C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Row–major layout of arrays

Case 1

\( n > \frac{M}{B}. \)

Analyze matrix \( B. \)
Assume LRU.

\( Q(n) = \Theta(n^3), \) since matrix \( B \) misses on every access.
Analysis of Cache Misses

```c
void Mult(double *C, double *A, double *B, int n) {
    for (int i=0; i < n; i++)
        for (int j=0; j < n; j++)
            for (int k=0; k < n; k++)
                C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

row–major layout of arrays

Case 2

\( M^{1/2} < n < cM/B \).

Analyze matrix B.

Assume LRU.

\[ Q(n) = n \cdot \Theta(n^2/B) = \Theta(n^3/B), \] since matrix B can exploit spatial locality.
TILING

SPEED LIMIT
PER ORDER OF 6.172
Tiled Matrix Multiplication

void Tiled_Mult(double *C, double *A, double *B, int n) {
    for (int i1=0; i1<n/s; i1+=s)
        for (int j1=0; j1<n/s; j1+=s)
            for (int k1=0; k1<n/s; k1+=s)
                for (int i=i1; i<i1+s&&i<n; i++)
                    for (int j=j1; j<j1+s&&j<n; j++)
                        for (int k=k1; k<k1+s&&k<n; k++)
                            C[i*n+j] += A[i*n+k] * B[k*n+j];
}

Analysis of work

- Work $W(n) = \Theta((n/s)^3(s^3))$
  $= \Theta(n^3)$.  

© 2013 Charles E. Leiserson and I-Ting Angelina Lee
Tiled Matrix Multiplication

void Tiled_Mult(double *C, double *A, double *B, int n) {
    for (int i1=0; i1<n/s; i1+=s)
        for (int j1=0; j1<n/s; j1+=s)
            for (int k1=0; k1<n/s; k1+=s)
                for (int i=i1; i<i1+s&&i<n; i++)
                    for (int j=j1; j<j1+s&&j<n; j++)
                        for (int k=k1; k<k1+s&&k<n; k++)
                            C[i*n+j] += A[i*n+k] * B[k*n+j];
}

Analysis of cache misses

- Tune $s$ so that the submatrices just fit into cache $\Rightarrow s = \Theta(M^{1/2})$.
- Tall-cache assumption implies $\Theta(s^2/B)$ misses per submatrix.
- $Q(n) = \Theta((n/s)^3(s^2/B)) = \Theta(n^3/BM^{1/2})$. Remember this!
- Optimal [HK81].
Tiled Matrix Multiplication

void Tiled_Mult(double *C, double *A, double *B, int n) {
    for (int i1=0; i1<n/s; i1+=s) {
        for (int j1=0; j1<n/s; j1+=s) {
            for (int k1=0; k1<n/s; k1+=s) {
                for (int i=i1; i<i1+s&&i<n; i++) {
                    for (int j=j1; j<j1+s&&j<n; j++) {
                        for (int k=k1; k<k1+s&&k<n; k++) {
                            C[i*n+j] += A[i*n+k] * B[k*n+j];
                        }
                    }
                }
            }
        }
    }
}

Analysis of cache misses

- Tune $s$ so that the submatrices just fit into cache $\Rightarrow s = \Theta(M^{1/2})$.
- Tall-cache assumption implies $\Theta(s^2/B)$ misses per submatrix.
- $Q(n) = \Theta((n/s)^3(s^2/B)) = \Theta(n^3/BM^{1/2})$. Remember this!
- Optimal [HK81].
Two-Level Cache

- Two “voodoo” tuning parameters $s$ and $t$.
- Multidimensional tuning optimization cannot be done with binary search.
Two-Level Cache

- Two “voodoo” tuning parameters $s$ and $t$.
- Multidimensional tuning optimization cannot be done with binary search.

```c
void Tiled_Mult2(double *C, double *A, double *B, int n) {
    for (int i2=0; i2<n/t; i2+=t)
        for (int j2=0; j2<n/t; j2+=t)
            for (int k2=0; k2<n/t; k2+=t)
                for (int i1=i2; i1<i2+t&&i1<n; i1+=s)
                    for (int j1=j2; j1<j2+t&&j1<n; j1+=s)
                        for (int k1=k2; k1<k2+t&&k1<n; k1+=s)
                            for (int i=i1; i<i1+s&&i<i2+t&&i<n; i++)
                                for (int j=j1; j<j1+s&&j<j2+t&&j<n; j++)
                                    for (int k=k1; k<k1+s&&k<k2+t&&k<n; k++)
                                        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```
Three-Level Cache

- Three “voodoo” tuning parameters.
- Twelve nested for loops.
- Multiprogrammed environment: Don’t know the effective cache size when other jobs are running ⇒ easy to mistune the parameters!
DIVIDE & CONQUER
Recursive Matrix Multiplication

Divide-and-conquer on $n \times n$ matrices.

$$
\begin{array}{cccc}
C_{11} & C_{12} & A_{11} & A_{12} \\
C_{21} & C_{22} & A_{21} & A_{22} \\
\end{array}
\times
\begin{array}{cccc}
B_{11} & B_{12} & A_{12} & A_{22} \\
B_{21} & B_{22} & A_{21} & A_{22} \\
\end{array}
= \\
\begin{array}{cccc}
A_{11}B_{11} & A_{11}B_{12} & A_{21}B_{11} & A_{21}B_{12} \\
A_{21}B_{11} & A_{21}B_{12} & A_{22}B_{21} & A_{22}B_{22} \\
\end{array}
+ \\
8 \text{ multiply-adds of } (n/2) \times (n/2) \text{ matrices.}
$$
Recursive Code

// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
               int n, int rowsize) {
    if (n == 1)
        C[0] += A[0] * B[0];
    else {
        int d11 = 0;
        int d12 = n/2;
        int d21 = (n/2) * rowsize;
        int d22 = (n/2) * (rowsize+1);

        Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
        Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
        Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
        Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
        Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
        Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
        Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
        Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
    }
}
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B, int n, int rowsize) {
    if (n == 1)
        C[0] += A[0] * B[0];
    else {
        int d11 = 0;
        int d12 = n/2;
        int d21 = (n/2) * rowsize;
        int d22 = (n/2) * (rowsize+1);

        Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
        Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
        Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
        Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
        Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
        Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
        Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
        Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
    } }

Analysis of Work

// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int n, int rowsize) {
    if (n == 1)
        C[0] += A[0] * B[0];
    else {
        int d11 = 0;
        int d12 = n/2;
        int d21 = (n/2) * rowsize;
        int d22 = (n/2) * (rowsize+1);

        Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
        Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
        Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
        Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
        Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
        Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
        Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
        Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
    }
}

\[ W(n) = 8W(n/2) + \Theta(1) = \Theta(n^3) \]
Analysis of Work

W(n) = 8W(n/2) + Θ(1)

recursion tree  W(n)
Analysis of Work

\[ W(n) = 8W(n/2) + \Theta(1) \]

recursion tree

\[ \frac{1}{8} \]

\[ W(n/2) \quad W(n/2) \quad \ldots \quad W(n/2) \]
Analysis of Work

\[ W(n) = 8W(n/2) + \Theta(1) \]
Analysis of Work

W(n) = 8W(n/2) + \Theta(1)

Note: Same work as looping versions. W(n) = \Theta(n^3)
Analysis of Cache Misses

// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
               int n, int rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int d11 = 0;
    int d12 = n/2;
    int d21 = (n/2) * rowsize;
    int d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
  }
}

Q(n) = \begin{cases} 
  \Theta(n^2/B) & \text{if } n^2 < cM \text{ for suff. small const } c \leq 1, \\
  8Q(n/2) + \Theta(1) & \text{otherwise}.
\end{cases}
Analysis of Cache Misses

\[
Q(n) = \begin{cases} 
\Theta(n^2/B) & \text{if } n^2 < cM \text{ for suff. small const } c \leq 1, \\
8Q(n/2) + \Theta(1) & \text{otherwise.}
\end{cases}
\]
Analysis of Cache Misses

\[ Q(n) = \begin{cases} 
\Theta(n^2/B) & \text{if } n^2 < cM \text{ for suff. small const } c \leq 1, \\
8Q(n/2) + \Theta(1) & \text{otherwise.} 
\end{cases} \]
Analysis of Cache Misses

\[ Q(n) = \begin{cases} 
\Theta(n^2/B) & \text{if } n^2 < cM \text{ for suff. small const } c \leq 1, \\
8Q(n/2) + \Theta(1) & \text{otherwise.} 
\end{cases} \]
Analysis of Cache Misses

\[
Q(n) = \begin{cases} 
\Theta(n^2/B) & \text{if } n^2 < cM \text{ for suff small const } c \leq 1, \\
8Q(n/2) + \Theta(1) & \text{otherwise.}
\end{cases}
\]

Geometric

\[
Q(n) = \Theta(n^3/BM^{1/2})
\]

Same cache misses as with tiling!
Efficient Cache–Oblivious Algorithms

- No voodoo tuning parameters.
- No explicit knowledge of caches.
- Passively autotune.
- Handle multilevel caches automatically.
- Good in multiprogrammed environments.

Matrix multiplication
The best cache–oblivious codes to date work on arbitrary rectangular matrices and perform binary splitting (instead of 8–way) on the largest of $i$, $j$, and $k$. 