Lecture 14
Caching and Cache-Efficient Algorithms

Julian Shun
Cache Hardware
Multicore Cache Hierarchy

- **Memory Controller**
- **Network**
- **LLC (L3)**
- **L2**
  - L1 data
  - L1 inst
- **P**
- **Level**
  - **Size**
  - **Assoc.**
  - **Latency (ns)**
  - Main: 128 GB, 50
  - LLC: 30 MB, 20
  - L2: 256 KB, 8
  - L1-d: 32 KB, 8
  - L1-i: 32 KB, 8

64 B cache blocks

© 2008–2018 by the MIT 6.172 Lecturers
A cache block can reside anywhere in the cache. To find a block in the cache, the entire cache must be searched for the tag. When the cache becomes full, a block must be evicted to make room for a new block. The replacement policy determines which block to evict.
Direct-Mapped Cache

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 searched.

<table>
<thead>
<tr>
<th>bits</th>
<th>tag</th>
<th>set</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w - \lg M$</td>
<td>$\lg(M/B)$</td>
<td>$\lg B$</td>
<td></td>
</tr>
</tbody>
</table>
Set–Associative Cache

A cache block’s set determines \( k \) possible cache locations.

To find a block in the cache, only the \( k \) locations of its set must be searched.

- \( w \)-bit address space
- \( \omega \)-bit address space
- Cache size \( M = 32 \).
- Line/block size \( B = 4 \).
- \( k = 2 \)-way associativity.

\[
\begin{array}{c|c|c|c}
\text{bits} & \text{tag} & \text{set} & \text{offset} \\
\hline
w - \lg(M/k) & \lg(M/kB) & \lg B \\
\end{array}
\]
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.
4096 columns of doubles
\[= 2^{15} \text{ bytes}\]

4096 rows

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

Conflict misses can be problematic for caches with limited associativity.

### Analysis
Look at a column of submatrix \( A \). The addresses of the elements are \( x, x+2^{15}, x+2 \cdot 2^{15}, \ldots, x+31 \cdot 2^{15} \). 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 $Q$
How Reasonable Are Ideal Caches?

“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

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

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

\[
\sum_{i=1}^{r} \frac{s_i}{B} + 2 = N/B + 2r
\]

\[
= N/B + 2rB)/B
\]

\[
\leq N/B + 2N/B
\]

\[
= 3N/B.
\]
Tall Caches

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

Example: Intel Xeon E5–2666 v3
- Cache-line length = 64 bytes.
- L1-cache size = 32 Kbytes.
What’s Wrong with Short Caches?

Tall-cache assumption

\[ B^2 < cM \]

for some sufficiently small constant \( c \leq 1 \).

An \( n \times n \) submatrix stored in row-major order may not fit in a short cache even if \( n^2 < cM \)!
Lemma. Suppose that an $n \times n$ submatrix $A$ is read into a tall cache satisfying $B^2 < cM$, where $c \leq 1$ is constant, and suppose that $cM \leq n^2 < M/3$. Then $A$ fits into cache, and the number of misses to read all $A$’s elements is at most $3n^2/B$.

Proof. We have $N = n^2$, $n = r = s_i$, $B \leq n = N/r$, and $N < M/3$. Thus, the Cache–Miss Lemma applies.
Cache Analysis of Matrix Multiplication
void Mult(double *C, double *A, double *B, int64_t n) {
    for (int64_t i=0; i < n; i++)
        for (int64_t j=0; j < n; j++)
            for (int64_t k=0; k < n; k++)
                C[i*n+j] += A[i*n+k] * B[k*n+j];
}

Analysis of work

$W(n) = \Theta(n^3)$. 
void Mult(double *C, double *A, double *B, int64_t n) {
    for (int64_t i=0; i < n; i++)
        for (int64_t j=0; j < n; j++)
            for (int64_t k=0; k < n; k++)
                C[i*n+j] += A[i*n+k] * B[k*n+j];
}

Assume row major and tall cache

Case 1
\[ n > cM/B. \]
Analyze matrix B.
Assume LRU.
\[ Q(n) = \Theta(n^3), \] since matrix B misses on every access.
void Mult(double *C, double *A, double *B, int64_t n) {
    for (int64_t i=0; i < n; i++)
        for (int64_t j=0; j < n; j++)
            for (int64_t k=0; k < n; k++)
                C[i*n+j] += A[i*n+k] * B[k*n+j];
}

Assume row major and tall cache

Case 2
$c' M^{1/2} < n < c M / B$.
Analyze matrix B.
Assume LRU.

\[ Q(n) = n \cdot \Theta(n^2 / B) = \Theta(n^3 / B) \], since
matrix B can exploit spatial locality.
void Mult(double *C, double *A, double *B, int64_t n) {
    for (int64_t i=0; i < n; i++)
        for (int64_t j=0; j < n; j++)
            for (int64_t k=0; k < n; k++)
                C[i*n+j] += A[i*n+k] * B[k*n+j];
}

Assume row major and tall cache

Case 3
n < c'M^{1/2}.
Analyze matrix B.
Assume LRU.
Q(n) = \Theta(n^2/B),
since everything fits in cache!
void Mult(double *C, double *A, double *B, int64_t n) {
    for (int64_t i=0; i < n; i++)
        for (int64_t k=0; k < n; k++)
            for (int64_t j=0; j < n; j++)
                C[i*n+j] += A[i*n+k] * B[k*n+j];
}

Assume row major and tall cache

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
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
    for (int64_t i1=0; i1<n/s; i1+=s)
        for (int64_t j1=0; j1<n/s; j1+=s)
            for (int64_t k1=0; k1<n/s; k1+=s)
                for (int64_t i=i1; i<i1+s && i<n; i++)
                    for (int64_t j=j1; j<j1+s && j<n; j++)
                        for (int64_t k=k1; k<k1+s && k<n; k++)
                            C[i*n+j] += A[i*n+k] * B[k*n+j];
}

Analysis of work

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

```c
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n; i1+=s)
    for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1; i<i1+s && i<n; i++)
          for (int64_t j=j1; j<j1+s && j<n; j++)
            for (int64_t 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 $\Rightarrow s = \Theta(M^{1/2})$.
- Submatrix Caching Lemma 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 $[HK81]$. 

© 2008–2018 by the MIT 6.172 Lecturers
Tiled Matrix Multiplication

```c
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
    for (int64_t i1=0; i1<n; i1+=s)
        for (int64_t j1=0; j1<n; j1+=s)
            for (int64_t k1=0; k1<n; k1+=s)
                for (int64_t i=i1; i<i1+s && i<n; i++)
                    for (int64_t j=j1; j<j1+s && j<n; j++)
                        for (int64_t 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 $\Rightarrow s = \Theta(M^{1/2})$. 
- Submatrix Caching Lemma implies $\Theta(s^2/B)$ misses per submatrix.
- $Q(n) = \Theta((n/s)^3(s^2/B))$
  $$= \Theta(n^3/(BM^{1/2})).$$  \textbf{Remember this!}
- Optimal \cite{HK81}.
Two-Level Cache

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

```c
void Tiled_Mult2(double *C, double *A, double *B, int64_t n) {
    for (int64_t i2=0; i2<n; i2+=s)
        for (int64_t j2=0; j2<n; j2+=s)
            for (int64_t k2=0; k2<n; k2+=s)
                for (int64_t i1=i2; i1<i2+s && i1<n; i1+=t)
                    for (int64_t j1=j2; j1<j2+s && j1<n; j1+=t)
                        for (int64_t k1=k2; k1<k2+s && k1<n; k1+=t)
                            for (int64_t i=i1; i<i1+s && i<i2+t && i<n; i++)
                                for (int64_t j=j1; j<j1+s && j<j2+t && j<n; j++)
                                    for (int64_t k=k1; k1<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}{c c}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{array}
= 
\begin{array}{c c}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{array}
\times
\begin{array}{c c}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{array}
= 
\begin{array}{c c}
A_{11}B_{11} & A_{11}B_{12} \\
A_{21}B_{11} & A_{21}B_{12}
\end{array}
+
\begin{array}{c c}
A_{12}B_{21} & A_{12}B_{22} \\
A_{22}B_{21} & A_{22}B_{22}
\end{array}

8 multiply-adds of $(n/2) \times (n/2)$ matrices.
Recursive Code

// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
               int64_t n, int64_t rowsize) {
    if (n == 1)
        C[0] += A[0] * B[0];
    else {
        int64_t d11 = 0;
        int64_t d12 = n/2;
        int64_t d21 = (n/2) * rowsize;
        int64_t 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,
              int64_t n, int64_t rowsize) {
    if (n == 1)
        C[0] += A[0] * B[0];
    else {
        int64_t d11 = 0;
        int64_t d12 = n/2;
        int64_t d21 = (n/2) * rowsize;
        int64_t 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,
              int64_t n, int64_t rowsize) {
    if (n == 1)
        C[0] += A[0] * B[0];
    else {
        int64_t d11 = 0;
        int64_t d12 = n/2;
        int64_t d21 = (n/2) * rowsize;
        int64_t 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) + Θ(1) = Θ(n^3)
Analysis of Work

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

recursion tree

\( W(n) \)
Analysis of Work

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

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

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

\[
\begin{align*}
\text{#leaves} &= 8^{\lg n} = n^{\lg 8} = n^3
\end{align*}
\]

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,
    int64_t n, int64_t rowsize) {
    if (n == 1)
        C[0] += A[0] * B[0];
    else {
        int64_t d11 = 0;
        int64_t d12 = n/2;
        int64_t d21 = (n/2) * rowsize;
        int64_t 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

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

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

recursion tree

\[ \begin{array}{cccc}
1 & 1 & \cdots & 1 \\
Q(n/4) & Q(n/4) & \cdots & Q(n/4) \\
8 & 1 & & 8
\end{array} \]
Analysis of Cache Misses

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

Same cache misses as with tiling!

Geometric recursion tree

\[ \text{#leaves} = 8^{\lg n - \frac{1}{2}\lg(cM)} = \Theta(n^3/\mathcal{M}^{3/2}). \]
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 $k$. 

© 2008–2018 by the MIT 6.172 Lecturers
Recursive Parallel Matrix Multiply

// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B, int64_t n, int64_t rowsize) {
    if (n == 1)
        C[0] += A[0] * B[0];
    else {
        int64_t d11 = 0;
        int64_t d12 = n/2;
        int64_t d21 = (n/2) * rowsize;
        int64_t d22 = (n/2) * (rowsize+1);

        cilk_spawn Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
        cilk_spawn Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
        cilk_spawn Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
        Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
        cilk_sync;
        cilk_spawn Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
        cilk_spawn Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
        cilk_spawn Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
        Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
        cilk_sync;
    }
}
Theorem. Let $Q_P$ be the number of cache misses in a deterministic Cilk computation when run on $P$ processors, each with a private cache of size $M$, and let $S_P$ be the number of successful steals during the computation. In the ideal-cache model, we have

$$Q_P = Q_1 + O(S_P M / B),$$

where $M$ is the cache size and $B$ is the size of a cache block.

Proof. After a worker steals a continuation, its cache is completely cold in the worst case. But after $M/B$ (cold) cache misses, its cache is identical to that in the serial execution. The same is true when a worker resumes a stolen subcomputation after a cilk_sync. The number of times these two situations can occur is at most $2S_P$. □

$S_P = O(PT_\infty)$ in expectation

MORAL: Minimizing cache misses in the serial elision essentially minimizes them in parallel executions.

© 2008–2018 by the MIT 6.172 Lecturers
Recursive Parallel Matrix Multiply

```c
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize)
{
    if (n == 1)
        C[0] += A[0] * B[0];
    else {
        int64_t d11 = 0;
        int64_t d12 = n/2;
        int64_t d21 = (n/2) * rowsize;
        int64_t d22 = (n/2) * (rowsize+1);

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

Span: \( T_\infty(n) = 2T_\infty(n/2) + \Theta(1) = \Theta(n) \)

Cache misses: \( Q_p = Q_1 + O(S_p M/B) \)  
\( = \Theta(n^3/BM^{1/2}) + O(Pn M/B) \)
• Associativity in caches
• Ideal cache model
• Cache-aware algorithms
  • Tiled matrix multiplication
• Cache-oblivious algorithms
  • Divide-and-conquer matrix multiplication
• Cache efficiency analysis in Homework 8