Cache Performance Metrics

**Miss Rate**
- Fraction of memory references not found in cache (misses / references)
- Typical numbers: 3-10% for L1; can be quite small (e.g., < 1%) for L2, depending on size, etc.

**Hit Time**
- Time to deliver a line in the cache to the processor (including time to determine whether the line is in the cache).
- Typical numbers: 1-3 clock cycles for L1; 5-12 clock cycles for L2.

**Miss Penalty**
- Additional time required because of a miss.
- Typically 100-300 cycles for main memory.
Repeated references to variables are good (temporal locality).
Stride-1 reference patterns are good (spatial locality).

**Examples:**
Assume cold cache, 4-byte words, 4 word cache blocks.

```c
int sumarrayrows( int a[M][N] )
{
    int i, j, sum = 0;
    for( i = 0; i < M; i++ )
        for( j = 0; j < N; j++ )
            sum += a[i][j];
    return sum;
}
```

Miss rate = $1/4 = 25\%$

```c
int sumarraycols( int a[M][N] )
{
    int i, j, sum = 0;
    for( j = 0; j < N; j++ )
        for( i = 0; i < M; i++ )
            sum += a[i][j];
    return sum;
}
```

Miss rate = $100\%$
The Memory Mountain

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache

Slopes of Spatial Locality
Ridges of Temporal Locality

read throughput (MB/s)

stride (words)

working set size (bytes)
Slice through the memory mountain with stride = 1. This illustrates read throughput with different caches and memory.
A Slope of Spatial Locality

Slice through memory mountain with size = 256KB. This shows cache block size.

![Graph showing read throughput (MB/s) vs stride (words)]
Major Cache Effects to Consider.

- Total cache size: Exploit temporal locality and keep the working set small (e.g., by using blocking).
- Block size: Exploit spatial locality.

Description

- Multiply $N \times N$ matrices.
- $O(N^3)$ total operations.
- Accesses:
  - N reads per source element
  - N values summed per destination (but may be held in register).

```c
/* ijk */
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0; // in reg
        for (k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```
Assume:

- Line size = 32B (big enough for 4 64-bit words)
- Matrix dimension $N$ is very large.
- We can approximate $1/N$ as 0.0.
- Cache is not even big enough to hold multiple rows.

**Analysis Method:** Look at access pattern of the inner loop.
C arrays are allocated in row-major order.
- Each row is allocated in contiguous memory locations.

Stepping through columns in one row:
```c
for (i = 0; i < N; i++)
    sum += a[j][i];
```
- This accesses successive elements.
- If block size $B > 4$ bytes, exploits spatial locality.
- Compulsory miss rate = 4 bytes / $B$.

Stepping through rows in one column:
```c
for (i = 0; i < N; i++)
    sum += a[i][i];
```
- Accesses distant elements.
- No spatial locality!
- Compulsory miss rate = 1 (i.e., 100%).
Matrix Multiplication (ijk)

```c
/* ijk */
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;  // in reg
        for (k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>
Matrix Multiplication (jik)

```c
/* ijk */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;    // in reg
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication ($kij$)

```c
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Misses per Inner Loop
Iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kji)

```c
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Summary of Matrix Multiplication

\[ \text{ijk (}& \text{ jik):} \]
- 2 loads, 0 stores
- misses / iteration = 1.25

\[ \text{kij (}& \text{ ikj):} \]
- 2 loads, 1 store
- misses / iteration = 0.5

\[ \text{ jki (}& \text{ kji):} \]
- 2 loads, 1 store
- misses / iteration = 2.0

Miss rates are important, but not perfect predictors of performance. Code scheduling matters, also.
Example: Blocked matrix multiplication

- “Block” (in this context) does not mean “cache block.”
- It means a sub-block within the structure (matrix).
- Example: \( N = 8 \); sub-block size = 4.

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
= 
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

Key idea: Sub-blocks (e.g., \( A_{xy} \)) can be treated just like scalars.

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}
\]

\[
C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++)
            for (j=jj; j < min(jj+bsize,n); j++) {
                sum = 0.0;
                for (k=kk; k < min(kk+bsize,n); k++) {
                    sum += a[i][k] * b[k][j];
                }
                c[i][j] += sum;
            }
    }
}

for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++)
            for (j=jj; j < min(jj+bsize,n); j++) {
                sum = 0.0;
                for (k=kk; k < min(kk+bsize,n); k++) {
                    sum += a[i][k] * b[k][j];
                }
                c[i][j] += sum;
            }
    }
}
for (i=0; i<n; i++) {
  for (j=jj; j < min(jj+bsize, n); j++) {
    sum = 0.0;
    for (k=kk; k < min(kk+bsize, n); k++) {
      sum += a[i][k] * b[k][j];
    }
    c[i][j] += sum;
  }
}

Innermost loop pair multiplies a $1 \times bsize$ sliver of A by a $bsize \times bsize$ block of B and accumulates into a $1 \times bsize$ sliver of C.

Loop over i steps through n row slivers of A and C, using same B.

row sliver accessed bsize times

block reused n times in succession

update successive elements of sliver
On a Pentium, blocking (bijk and bikj) improves performance by a factor of two over the unblocked versions (ijk and jik).

The result is relatively insensitive to array size.
The programmer can optimize for cache performance.
- How data structures are organized.
- How data are accessed (e.g., nested loop structure).
- Blocking is a general technique.

All systems favor “cache friendly code.”
- Getting absolute optimum performance is very platform specific.
- Involves cache sizes, line sizes, associativities, etc.
- Can get most advantage with generic code.
- Keep working set reasonably small (temporal locality).
- Use small strides (spatial locality).