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 (16-byte) 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%
Why would performance drop as the working set gets very small?
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.

The diagram shows the read throughput (MB/s) as a function of stride (words) for different cache blocks labeled s1 to s16. The x-axis represents the stride in words, and the y-axis represents the read throughput in MB/s. The diagram indicates that one access per cache line is achieved, as marked by the arrows.
Anomaly in Memory Mountain

Why does the memory mountain drop off at the back? Prof. Warren Hunt told me: “When I looked into this issue, I didn’t come to a clean resolution. Perhaps the dropoff is a measurement anomaly; the times are so short in comparison to the measurement costs that it appears that the performance is degrading.”
Matrix Multiplication Example

Major Cache Effects to Consider.
- Total cache size: Exploit temporal locality and keep the working set small
- 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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication \((jik)\)

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

\[
\begin{align*}
/* \text{kij} */ \\
\text{for} \ (k=0; \ k<n; \ k++) \ { \\
\indent \text{for} \ (i=0; \ i<n; \ i++) \ { \\
\indent \indent r = a[i][k]; \\
\indent \text{for} \ (j=0; \ j<n; \ j++) \\
\indent \indent \text{c}[i][j] += r * b[k][j]; \\
\indent } \\
\} \\
\}
\end{align*}
\]

Misses per Inner Loop Iteration:

\[
\begin{array}{ccc}
A & B & C \\
0.0 & 0.25 & 0.25 \\
\end{array}
\]
Matrix Multiplication (ikj)

```c
/* 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)

/* 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>Misses</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.
The programmer can optimize for cache performance.

- How data structures are organized.
- How data are accessed (e.g., nested loop structure).

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).