The Memory Hierarchy

Simple model

CPU
execute instructions

instructions + data

Memory System
linear array of bytes.

↑ Smaller, faster, and costlier.

L0: Registers
L1 cache
L2 cache
L3 cache
Main Memory
Local disks
Remote Secondary Storage

L6
<table>
<thead>
<tr>
<th>Memory Type</th>
<th>No. of CPU Cycles to Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU register</td>
<td>1</td>
</tr>
<tr>
<td>Cache</td>
<td>1 - 30 cycles</td>
</tr>
<tr>
<td>Main Memory</td>
<td>50 - 200 cycles</td>
</tr>
<tr>
<td>Disk</td>
<td>1 - 10 million cycles</td>
</tr>
</tbody>
</table>

Locality

Programs with good locality tend to access the same set of data items over and over again, or they tend to access sets of nearby data items.

Programs with good locality access most of their data items here.
Principle of locality / Locality of reference:
Same values or related storage locations are frequently accessed.

Locality

Temporal Locality
A memory location that is referenced once is likely to be referenced again multiple times.

Spatial Locality
If a memory location is referenced once, then the program is likely to reference a nearby memory location in the near future.

Why is this important?
Programs with good locality run faster than programs with poor locality.

Idea of locality is used in:

1. Hardware
   - Cache memories vs. main memory
   - Main memory vs. hard disk

2. OS
   - Web servers vs. cache
   - Databases vs. buffer pool

3. Appln. programs
Locality of References to Program Data

A program to sum the contents of an array:

```c
int sum_array(int a[], int n) {
    int i, sum = 0;
    for (i = 0; i < n; i++) {
        sum += a[i];
    }
    return sum;
}
```

```
a: 0 1 2
    3 7 9

sum: 19
a[2]: 9
a[1]: 7
a[0]: 3
```

```
0x200
0x108
0x104
0x100
```
Does this function have good locality?

Reference pattern for each variable.

1. **Sum variable**
   - Referenced once in each loop iteration.
     ⇒ Temporal Locality w.r.t. sum.
   - There is **NO** spatial locality w.r.t. sum.
     ⇒ sum is a scalar. (i.e. a single variable).

2. **Variable 'a'**
   - Elements of array 'a' are read sequentially one after the others, in the order they are stored in memory.
     ⇒ Spatial Locality w.r.t. variable 'a'.
   - There is **NO** temporal locality w.r.t. a.
     ⇒ each array element is accessed exactly once.

⇒ sum array() function has good locality!
Sequential Reference Pattern (or) Stride-1 Reference pattern

* Each element of the array is accessed sequentially.

Access Order: 1 2 3 4 5 6 7 8 9 10

Stride-K Reference pattern

* Visiting every $k^{th}$ element of a contiguous vector.

In stride-1 reference spatial locality. Stride-K reference patterns exhibit excellent spatial locality.

Stride-K reference patterns don't exhibit
2-d arrays in C

```c
int a[2][3];
```

```
\begin{tabular}{c|c|c}
  0 & 1 & 2 \\
\hline
  a_{00} & a_{01} & a_{02} \\
  a_{10} & a_{11} & a_{12} \\
\end{tabular}
```

Addr: 0 4 8 12 16 20

Contents: a_{00} a_{01} a_{02} a_{10} a_{11} a_{12}

Sum the 2-d array row wise

```c
int sum_array_rows (int a[2][3], int m, int n) {
    int i, j, sum = 0;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            sum += a[i][j];
        }
    }
    return sum;
}
```
Row-wise sum of 2-d array

\[ \text{enjoys good spatial locality} \]

The array is referenced in the same row-major order in which it is stored in memory.

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>a_{00}</td>
<td>a_{01}</td>
<td>a_{02}</td>
<td>a_{10}</td>
<td>a_{11}</td>
<td>a_{12}</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Column-wise sum of 2-d array:

```c
int sum_array_cols (int a[3][3], int m, int n)
{
    int i, j, sum = 0;
    for (j = 0; j < n; j++)
        for (i = 0; i < m; i++)
            sum += a[i][j];
    return sum;
}
```
Column-wise sum of 2-d array
→ suffers from poor spatial locality.
("array is scanned col-wise but it is organized row-wise in memory.

: The result is a stride-N reference pattern.

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>$a_{00}$</td>
<td>$a_{01}$</td>
<td>$a_{02}$</td>
<td>$a_{10}$</td>
<td>$a_{11}$</td>
<td>$a_{12}$</td>
</tr>
<tr>
<td>Access Order</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
eq. Instructions in the body of the for loop.

\[ \text{sum_array} : \ldots \]

8048400: eb 18

8048402: 8b 45 fc

\[ \text{mov} \ -0x4(\text{\_ebp}), \text{\_eax} \]

804841a: 8b 45 fc

\[ \text{mov} \ -0x4(\text{\_ebp}), \text{\_eax} \]

804841d: 3b 45 0c

\[ \text{cmp} \ 0xc(\text{\_ebp}), \text{\_eax} \]

8048420: 7c e0

\[ \text{jl} \ 8048402 \]

1. Instruction in loop body executed in sequential memory.

\[ \Rightarrow \text{loop enjoys good spatial locality.} \]

2. The loop is executed multiple times

\[ \Rightarrow \text{loop also enjoys good temporal locality.} \]
Caching in the Memory Hierarchy

Cache (in general) - small, fast storage device that acts as a staging area for the data objects stored in a larger, slower device.

The process of using a cache = caching!

Local disk - cache for web pages retrieved from Web Servers.

Main memory - cache for data on the local disks.

Cache memories - main memory

CPU registers - CPU caches.

Level $k$: faster and smaller storage device

Level $k+1$: slower and larger storage device

Storage device at level $k$ is a cache for the storage device at level $k+1$. 
Basic principle of caching

Level K:

\[
\begin{array}{cccc}
4 & 9 & 14 & 3 \\
\end{array}
\]

Data is copied between levels in block-sized transfer units.

Level K+1:

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
\end{array}
\]

Larger, slower, cheaper storage.

Transfer units

\[
\begin{align*}
L_0 & \leftrightarrow L_1 \\
L_1 & \leftrightarrow L_2 \\
L_2 & \leftrightarrow L_3 \\
L_3 & \leftrightarrow L_4 \\
L_4 & \leftrightarrow L_5 \\
\end{align*}
\]

Size of data transferred

\[
\begin{align*}
\rightarrow 1 \text{ word blocks (i.e. } 1 \text{ block = 1 word)} \\
\rightarrow 8 \text{ to } 16 \text{ words in a block} \\
\rightarrow > 16 \text{ words in a block}
\end{align*}
\]
The code snippet appears to be a fragment of a program, possibly in assembly language. The diagram shows a flowchart or a block diagram with various instructions and labels like `ret`, `jmp`, `addl`, etc. It seems to be related to a loop or a procedure with conditional checks and arithmetic operations. The exact context and purpose would require further analysis of the surrounding code or documentation.