Locality

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - **Temporal locality**: Recently referenced items are likely to be referenced in the near future.
  - **Spatial locality**: Items with nearby addresses tend to be referenced close together in time.

Locality Example:

- **Data**
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference `sum` each iteration: **Temporal locality**

- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Locality Example:

**Claim**: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

**Question**: Does this function have good locality?

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

Locality Example:

**Question**: Does this function have good locality?

```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;
}
```

Locality Example:

**Question**: Can you permute the loops so that the function scans the 3-d array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sumarray3d(int a[M][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[i][j][k];
    return sum;
}
```
Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte and have less capacity.
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

An Example Memory Hierarchy

Caches

- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
  - Net effect: A large pool of memory that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Caching in a Memory Hierarchy
General Caching Concepts

- Program needs object d, which is stored in some block b.
- Cache hit
  - Program finds b in the cache at level k. E.g., block 14.
- Cache miss
  - b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
  - If level k cache is full, then some current block must be replaced (evicted). Which one is the “victim”?
  - Placement policy: where can the new block go? E.g., b mod 4
  - Replacement policy: which block should be evicted? E.g., LRU

Types of cache misses:
- Cold (compulsory) miss
  - Cold misses occur because the cache is empty.
- Conflict miss
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
  - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k+1.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
  - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Examples of Caching in the Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What Cached</th>
<th>Where Cached</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-byte word</td>
<td>CPU registers</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32-byte block</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>32-byte block</td>
<td>On-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware+OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>

Examples of Caching in the Hierarchy

Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in L1, then in L2, then in main memory.
- Typical bus structure:
Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is a 4-byte block.
The transfer unit between the cache and main memory is a 4-word block (16 bytes).
The small fast L1 cache has room for two 4-word blocks.
The big slow main memory has room for many 4-word blocks.

General Org of a Cache Memory

Cache is an array of sets.
Each set contains one or more lines.
Each line holds a block of data.

Addressing Caches

Address A:
\[ \text{Address A: } t \text{ bits } s \text{ bits } b \text{ bits} \]

Direct-Mapped Cache

- Simplest kind of cache
- Characterized by exactly one line per set.
### Accessing Direct-Mapped Caches

- **Set selection**
  - Use the set index bits to determine the set of interest.

```
<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>
```

- **Line matching and word selection**
  - **Line matching**: Find a valid line in the selected set with a matching tag
  - **Word selection**: Then extract the word

```
<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>
```

### Direct-Mapped Cache Simulation

- **M=16 byte addresses, B=2 bytes/block, S=4 sets, E=1 entry/set**
- **Address trace (reads):** 0 [0000], 1 [0001], 13 [1101], 8 [1000], 0 [0000]

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 [0000]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 [0001]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13 [1101]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8 [1000]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 [0000]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

### Why Use Middle Bits as Index?

- **High-Order Bit Indexing**
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality
- **Middle-Order Bit Indexing**
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 [0000]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 [0001]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 [0001]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8 [1000]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 [0000]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Set Associative Caches

- Characterized by more than one line per set

Set 0:
- Valid
- Tag
- Cache block

$2^{E} = 2$ lines per set

Set 1:
- Valid
- Tag
- Cache block

Set $S-1$:
- Valid
- Tag
- Cache block

Accessing Set Associative Caches

- Set selection
  - Identical to direct-mapped cache

- Line matching and word selection
  - Must compare the tag in each valid line in the selected set.

Selected set
- Tag set index block offset

Set 0:
- Valid
- Tag
- Cache block

Set 1:
- Valid
- Tag
- Cache block

Set $S-1$:
- Valid
- Tag
- Cache block

Accessing Set Associative Caches

- (1) The valid bit must be set.
- (2) The tag bits in one of the cache lines must match the tag bits in the address.
- (3) If (1) and (2), then cache hit, and block offset selects starting byte.

Cache Organization

- Location – how do you locate a block
- Placement – where is a block placed
- Re-placement – which block do you replace
  - Least recently used (LRU)
  - FIFO
  - Random
- Write policy – what happens on a write
  - Write back
  - Write through no write allocate
  - Write through write allocate
Multi-Level Caches

- Options: separate data and instruction caches, or a unified cache

Processor
- L1 d-cache
- L1 i-cache
- Unified L2 Cache
- Memory
- disk

Size: 200 B 8-64 KB 1-4 MB SRAM 128 MB DRAM 30 GB
Speed: 3 ns 3 ns 6 ns 60 ns 8 ns
$/Mbyte: $100/MB $1.50/MB $0.05/MB
Line size: 8 B 32 B 32 B 8 KB

large, slower, cheaper

Intel Pentium Cache Hierarchy

- L1 Data
  - 1 cycle latency
  - 16 KB
  - 4-way assoc
  - Write-through
  - 32B lines
- L1 Instruction
  - 16 KB, 4-way
  - 32B lines
- L2 Unified
  - 128 KB–2 MB
  - 4-way assoc
  - Write-back
  - Write allocate
  - 32B lines
- Main Memory
  - Up to 4 GB

Other Optimizations

- Write buffers
- Prefetching

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 (includes time to determine whether the line is in the cache)
  - Typical numbers:
    - 1-2 clock cycles for L1
    - 8-15 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - Typically 25-100 cycles for main memory