Memory Hierarchy: Cache

Memory hierarchy
Cache basics
Locality
Cache organization
Cache-aware programming

How does execution time grow with SIZE?

```java
int[] array = new int[SIZE];
fillArrayRandomly(array);
int s = 0;
for (int i = 0; i < 200000; i++) {
    for (int j = 0; j < SIZE; j++) {
        s += array[j];
    }
}
```

reality beyond O(...)
Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus bandwidth evolved much slower

Core 2 Duo:
Can process at least 256 Bytes/cycle

Core 2 Duo:
Bandwidth 2 Bytes/cycle
Latency 100 cycles

Solution: caches

Cache

English definition:
n. a hidden storage space for provisions, weapons, or treasures
v. to store away in hiding for future use

CS definition:
n. a computer memory with short access time used to store frequently or recently used instructions or data
v. to store [data/instructions] temporarily for later quick retrieval

Also used more broadly in CS: software caches, file caches, etc.

General Cache Mechanics

Cache

Data is copied in block-sized transfer units

Memory

Smaller, faster, more expensive memory cache a subset of the blocks (a.k.a. lines)

Larger, slower, cheaper memory viewed as partitioned into "blocks" or "lines"

General Cache Concepts: Hit

Request: 14

Request data in block b.

Block b is in cache:
Hit
General Cache Concepts: Miss

Request data in block b.
Block b is not in cache: Miss!

Fetch block b from memory.

Store block b in cache.

• Placement policy: determines where b goes
• Replacement policy: determines which block gets evicted (victim)

Locality: why caches work

Locality:
Programs tend to use data and instructions at addresses near or equal to those they have used recently.

Temporal locality:
Recently referenced items are likely to be referenced again in the near future.

Spatial locality:
Items with nearby addresses are likely to be referenced close together in time.

How do caches exploit temporal and spatial locality?

Example: Locality?

```c
int sum_array_rows(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;
}
```

Data:
Temporal: sum referenced in each iteration
Spatial: array a[] accessed in stride-2 pattern

Instructions:
Temporal: cycle through loop repeatedly
Spatial: reference instructions in sequence

Assessing locality in code is an important programming skill.
Locality Example #2

```c
int sum_array_cols(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;
}
```

What is wrong with this code? How can it be fixed?

Locality Example #3

```c
int sum_array_3d(int a[M][N][N]) {
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < M; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

Cost of Cache Misses

Huge difference between a hit and a miss
Could be 100x, if just L1 and main memory

99% hits could be twice as good as 97%. How?
Cache hit time of 1 cycle, miss penalty of 100 cycles

Mean access time:
97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

This is why “miss rate” is used instead of “hit rate”

Cache Performance Metrics

Miss Rate
Fraction of memory accesses to data not in cache (misses / accesses)
Typically: 3% - 10% for L1; maybe < 1% for L2, depending on size, etc.

Hit Time
Time to find and deliver a line in the cache to the processor.
Typically: 1 - 2 clock cycles for L1; 5 - 20 clock cycles for L2

Miss Penalty
Additional time required on cache miss = main memory access time
Typically 50 - 200 cycles for L2 (trend: increasing)
(mostly) fundamental/enduring properties of memory/storage:
Faster = smaller = costlier.
Speed gaps are widening: registers ⇐ cache ⇐ DRAM ⇐ disk.
Well-written programs tend to exhibit good locality.

Memory hierarchy:
Faster, smaller device is a cache for larger, slower device at next level.

Why do memory hierarchies work?
Locality keeps most accesses in faster, smaller, costlier level.
Capacity in slower, larger, cheaper level can store all data and serve rare non-local requests.

Big Idea
The memory hierarchy creates a large pool of storage 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.
Where should we put data in the cache?

Direct-mapped cache

cache block from address \( a \) in slot \( a \mod \#slots \)

Tags record what is cached.

tag = rest of address bits not used for index

What’s a cache block? (or cache line)

typical block/line sizes: 32 bytes, 64 bytes
A puzzle.

Cache starts empty
Access (addr, hit/miss) stream:

(10, miss), (11, hit), (12, miss)

Wouldn’t it be nice if we could store a block in any slot in the cache?

Problems with direct-mapped caches?

direct mapped:
Each memory address maps to exactly one cache index.

What happens when accessing 2, 6, 2, 6, 2, ...?

conflict

Wouldn’t it be nice if we could store a block in any slot in the cache?

Associativity

Each block maps to exactly one set, may be cached in any slot in that set.

<table>
<thead>
<tr>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 sets, 1 block each</td>
<td>4 sets, 2 blocks each</td>
<td>2 sets, 4 blocks each</td>
<td>1 set, 8 blocks</td>
</tr>
</tbody>
</table>

direct mapped  | 2-way  | 4-way  | 8-way  |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where does data get cached?

m-bit Address  | (m-k-n) bits | k bits | n-bit Block Offset
| Set | Index | Offset |

11/21/14
What’s a cache block? (or cache line)

Where does data get cached?

Example placement in set-associative caches

Block replacement

Block/line size = 16 bytes

Where would the byte at address 13 (1101) be stored?

Example: direct-mapped 2^2-block cache with 2^4 bytes per block.

Where could data from address 0x1833 be placed?

1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each

direct mapped
Another puzzle.

Cache starts empty
Access (addr, hit/miss) stream
(10, miss); (12, miss); (10, miss)

Cache Read

General Cache Organization (S, E, B)

Example: Direct-Mapped Cache (E = 1)
Example: Direct-Mapped Cache (E = 1)
Direct-mapped: One line per set
Assume: cache block size 8 bytes

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>yes</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>yes</td>
<td>7</td>
</tr>
</tbody>
</table>

Address of int:

Block offset:

Example (for E = 1)

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Example (for E = 1)

```c
int dotprod(int x[8], int y[8])
{
    int sum = 0;
    int i;
    for (i = 0; i < 8; i++)
        sum += x[i] * y[i];
    return sum;
}
```
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of int:

<table>
<thead>
<tr>
<th>E</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
</tr>
<tr>
<td>1</td>
<td>v6</td>
<td>v7</td>
<td>v8</td>
</tr>
<tr>
<td>2</td>
<td>v9</td>
<td>v10</td>
<td>v11</td>
</tr>
</tbody>
</table>

E = 2:
Two lines per set

Assume:
cache block size 8 bytes

Example (for E = 2)

```cpp
float dotprod(float x[], float y[])
{
    float sum = 0;
    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```

If x and y have aligned starting addresses, e.g., &x[0] = 0, &y[0] = 128, can still fit both because two lines in each set...
Types of Cache Misses

Cold (compulsory) miss
- first access to a block

Conflict miss
- cache is large enough to hold all needed blocks, but multiple blocks map to same slot
  - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
  - increasing associativity can reduce conflict misses

Capacity miss
- working set of active cache blocks is larger than the cache

What about writes?

Multiple copies of data exist:
L1, L2, possibly L3, main memory

What is the main problem with that?

Write-hit policy
- Write-through: write immediately to memory, all caches in between
- Write-back: defer write to memory until line is evicted (replaced)
  - Need a dirty bit to indicate if line is different from memory or not

Write-miss policy
- Write-allocate: load into cache, update line in cache
  - Good if more nearby writes or reads follow
- No-write-allocate: just write immediately to memory

Typical caches:
- Write-back + Write-allocate, usually
- Write-through + No-write-allocate, occasionally

Write-back, write-allocate example

```
li $t0, 0xCAFE
la $t1, T
sw $t0, 0($t1)
```

Cache: 0xBEEF
Memory: 0xCAFE

Dirty bit
Write-back, write-allocate example

```assembly
li $t0, 0xA0C3
la $t8, T
sw $t0, 0($t8)  

li $t0, 0x0BEF
sw $t0, 0($t1)
```

### Memory

- **$t0**: 0xA0C3
- **$t1**: 0x0BEF

### Cache

- **dirty bit**: 1

#### Back to the Core i7 to look at ways

Processor package:
- **Core 0**:
  - L1 d-cache (shared by all cores)
  - L1 i-cache (shared by all cores)
- **Core 3**:
  - L1 d-cache
  - L1 i-cache
  - L2 unified cache
  - L3 unified cache

#### Aside: software caches

**Examples**
- File system buffer caches, web browser caches, database caches, network CDN caches, etc.

**Some design differences**
- Almost always fully-associative
  - no placement restrictions
- Index structures like hash tables are common (for placement)
- Often use complex replacement policies
  - misses are very expensive when disk or network involved
  - worth thousands of cycles to avoid them
- Not necessarily constrained to single "block" transfers
  - may fetch or write-back in larger units, opportunistically
Cache-Friendly Code

Locality, locality, locality.

Programmer can optimize for cache performance
- How data structures are organized
- How data are accessed
- Nested loop structure
- Blocking is a general technique

All systems favor “cache-friendly code”
- Getting absolute optimum performance is hardware-specific
- Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
- Keep working set reasonably small (temporal locality)
- Use small strides (spatial locality)
- Focus on inner loop code

The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip

Working set size (bytes)
Stride (x8 bytes)
Read throughput (MB/s)