Memory Hierarchy: Caching
The memory subsystem
Memory Locality

• Memory hierarchies take advantage of *memory locality*.

• *Memory locality* is the principle that future memory accesses are *near* past accesses.

• Memories take advantage of two types of locality
  – -- near in time => we will often access the same data again very soon
  – -- near in space/distance => our next access is often very close to our last access (or recent accesses).

(this sequence of addresses exhibits both temporal and spatial locality)
1,2,3,1,2,3,8,8,47,9,10,8,8...
Locality and Caching

• Memory hierarchies exploit locality by caching (keeping close to the processor) data likely to be used again.
• This is done because we can build large, slow memories and small, fast memories, but we can’t build large, fast memories.
• If it works, we get the illusion of SRAM access time with disk capacity

SRAM access times are 0.5-5ns at cost of $4k to $10k per Gbyte.
DRAM access times are 50-70ns at cost of $100 to $200 per Gbyte.
Disk access times are 5 to 20 million ns at cost of $0.50 to $2 per Gbyte.

(source: text)
A typical memory hierarchy

- CPU
- on-chip cache
- off-chip cache
- main memory
- disk

small, expensive $/bit

big, cheap $/bit

so then where is my program and data??
Cache Fundamentals

- **cache hit** -- an access where the data is found in the cache.
- **cache miss** -- an access which isn’t
- **hit time** -- time to access the cache
- **miss penalty** -- time to move data from further level to closer, then to cpu
- **hit ratio** -- percentage of time the data is found in the cache
- **miss ratio** -- (1 - hit ratio)
Cache Fundamentals, cont.

- **cache block size** or **cache line size**— the amount of data that gets transferred on a cache miss.
- **instruction cache** -- cache that only holds instructions.
- **data cache** -- cache that only caches data.
- **unified cache** -- cache that holds both.
Caching Issues

On a memory access -
• How do I know if this is a hit or miss?

On a cache miss -
• where to put the new data?
• what data to throw out?
• how to remember what data this is?
A simple cache

- A cache that can put a line of data anywhere is called ________________.
- The most popular replacement strategy is LRU ( )

The tag identifies the address of the cached data

<table>
<thead>
<tr>
<th>address string:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>12</td>
<td>00001100</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00001100</td>
</tr>
<tr>
<td>24</td>
<td>00011000</td>
</tr>
<tr>
<td>12</td>
<td>00001100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
</tbody>
</table>
A simpler cache

• A cache that can put a line of data in exactly one place is called ________________.

• Advantages/disadvantages vs. fully-associative?

address string:
4 00000100
8 00001000
12 00000100
4 00000100
8 00001000
20 00010100
4 00000100
20 00010100
24 00011000
12 00001100
8 00001000
4 00000100

4 entries, each block holds one word, each word in memory maps to exactly one cache location.
A set-associative cache

<table>
<thead>
<tr>
<th>address string</th>
<th>00000100</th>
<th>00001000</th>
<th>00001100</th>
<th>00010100</th>
<th>00001000</th>
<th>00011000</th>
<th>00001000</th>
<th>00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>00000100</td>
<td>00001100</td>
<td>00000100</td>
<td>00001010</td>
<td>00000100</td>
<td>00001010</td>
<td>00001000</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00000100</td>
<td>00000100</td>
<td>00001100</td>
<td>00000100</td>
<td>00001000</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td>12</td>
<td>00000100</td>
<td>00000100</td>
<td>00001100</td>
<td>00000100</td>
<td>00001000</td>
<td>00000100</td>
<td>00000100</td>
<td>00000100</td>
</tr>
</tbody>
</table>

- A cache that can put a line of data in exactly \( n \) places is called **n-way set-associative**.
- The cache lines/blocks that share the same index are a cache **set**.
Longer Cache Blocks

- Large cache blocks take advantage of *spatial locality*.
- Too large of a block size can waste cache space.
- Longer cache blocks require less tag space.
Block Size and Miss Rate

![Graph showing the relationship between block size and miss rate.](image)

- **1 KB**
- **8 KB**
- **16 KB**
- **64 KB**
- **256 KB**
Cache Parameters

Cache size = Number of sets * block size * associativity

- 128 blocks, 32-byte block size, direct mapped, size = ?

- 128 KB cache, 64-byte blocks, 512 sets, associativity = ?
Handling a Cache Access

1. Use index and tag to access cache and determine hit/miss.
2. If hit, return requested data.
3. If miss, select a cache block to be replaced, and access memory or next lower cache (possibly stalling the processor).
   - load entire missed cache line into cache
   - return requested data to CPU (or higher cache)
4. If next lower memory is a cache, goto step 1 for that cache.
Accessing a Sample Cache

- 64 KB cache, direct-mapped, 32-byte cache block size

64 KB / 32 bytes = 2 K cache blocks/sets
Accessing a Sample Cache

- 32 KB cache, 2-way set-associative, 16-byte block size

```
31 30 29 28 27 ........... 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

```
tag index

valid tag data valid tag data
```

```
32 KB / 16 bytes / 2 = 1 K cache sets
```

```
hit/miss

CSE 141, S2'06
Jeff Brown
```
Associative Caches

• Higher hit rates, but...

• longer access time (longer to determine hit/miss, more muxing of outputs)

• more space (longer tags)
  – 16 KB, 16-byte blocks, dm, tag = ?
  – 16 KB, 16-byte blocks, 4-way, tag = ?
Dealing with Stores

- Stores must be handled differently than loads, because...
  - they don’t necessarily require the CPU to stall.
  - they change the content of cache/memory (creating memory consistency issues)
  - may require a                 and a store to complete
Policy decisions for stores

• Keep memory and cache identical?
  – => all writes go to both cache and main memory
  – => writes go only to cache. Modified cache lines are written back to memory when the line is replaced.

• Make room in cache for store miss?
  – write-allocate => on a store miss, bring written line into the cache
  – write-around => on a store miss, ignore cache
Dealing with stores

• On a store hit, write the new data to cache. In a write-through cache, write the data immediately to memory. In a write-back cache, mark the line as dirty.

• On a store miss, initiate a cache block load from memory for a write-allocate cache. Write directly to memory for a write-around cache.

• On any kind of cache miss in a write-back cache, if the line to be replaced in the cache is dirty, write it back to memory.
Cache Performance

CPI = BCPI + MCPI

- BCPI = base CPI, which means the CPI assuming perfect memory (BCPI = peak CPI + PSPI + BSPI)
  - PSPI => pipeline stalls per instruction
  - BSPI => branch hazard stalls per instruction
- MCPI = the memory CPI, the number of cycles (per instruction) the processor is stalled waiting for memory.

MCPI = accesses/instruction * miss rate * miss penalty

- this assumes we stall the pipeline on both read and write misses, that the miss penalty is the same for both, that cache hits require no stalls.
- If the miss penalty or miss rate is different for Inst cache and data cache (common case), then

MCPI = I$ accesses/inst*I$MR*I$MP + D$ acc/inst*D$MR*D$MP
Cache Performance

- Instruction cache miss rate of 4%, data cache miss rate of 9%, BCPI = 1.0, 20% of instructions are loads and stores, miss penalty = 12 cycles, CPI = ?
Cache Performance

• Unified cache, 25% of instructions are loads and stores, BCPI = 1.2, miss penalty of 10 cycles. If we improve the miss rate from 10% to 4% (e.g. with a larger cache), how much do we improve performance?
Cache Performance

- BCPI = 1, miss rate of 8% overall, 20% loads, miss penalty 20 cycles, never stalls on stores. What is the speedup from doubling the cpu clock rate?
Example -- DEC Alpha 21164 Caches

- ICache and DCache -- 8 KB, DM, 32-byte lines
- L2 cache -- 96 KB, ?-way SA, 32-byte lines
- L3 cache -- 1 MB, DM, 32-byte lines
The data that gets moved into the cache on a miss are all data whose addresses share the same tag and index (regardless of which data gets accessed first).

This results in:
- no overlap of cache lines
- easy mapping of addresses to cache lines (no additions)
- data at address X always being present in the same location in the cache block (at byte $X \mod \text{blocksize}$) if it is there at all.

Think of main memory as organized into cache-line sized pieces (because in reality, it is!).
Cache Associativity

Miss rate

15%
12%
9%
6%
3%
0%

One-way
Two-way
Four-way
Eight-way

Associativity

1 KB
2 KB
4 KB
8 KB
16 KB
32 KB
64 KB
128 KB
Three types of cache misses

- **Compulsory (or cold-start) misses**
  - first access to the data.

- **Capacity misses**
  - we missed only because the cache isn’t big enough.

- **Conflict misses**
  - we missed because the data maps to the same line as other data that forced it out of the cache.
So, then, how do we decrease...

• Compulsory misses?
• Capacity misses?
• Conflict misses?
LRU replacement algorithms

- only needed for associative caches
- requires one bit for 2-way set-associative, 8 bits for 4-way, 24 bits for 8-way.
- can be emulated with \( \log n \) bits (NMRU)
- can be emulated with \( \text{use} \) bits for highly associative caches (like page tables)
- However, for most caches (eg, associativity \( \leq 8 \)), LRU is calculated exactly.
Caches in Current Processors

- A few years ago, they were DM closest to CPU, associative further away (this is less true today).
- split I and D close to the processor (for throughput rather than miss rate), unified further away.
- write-through and write-back both common, but never write-through all the way to memory.
- 32-byte cache lines common (but getting larger – 64, 128)

- Non-blocking
  - processor doesn’t stall on a miss, but only on the use of a miss (if even then)
  - this means the cache must be able to handle multiple outstanding accesses.
Key Points

- Caches give illusion of a large, cheap memory with the access time of a fast, expensive memory.
- Caches take advantage of memory locality, specifically temporal locality and spatial locality.
- Cache design presents many options (block size, cache size, associativity, write policy) that an architect must combine to minimize miss rate and access time to maximize performance.