CS 261
Fall 2019

Mike Lam, Professor

Caching
(get it??)
Topics

- Caching
- Cache implementations
- Cache policies
- Cache performance
- Performance improvement strategies
Motivation

- Caching is ubiquitous in modern computing:
  - L1-L3 memory
  - TLB and virtual memory (next week)
  - Disk controller buffers
  - Network controller buffers
  - Browser caches
  - Content delivery networks
Caching

- A **cache** is a small, fast memory that acts as a buffer or staging area for a larger, slower memory.
  - Fundamental CS system design concept
  - Data is transferred in **blocks** or **lines**
  - Slower caches use larger block sizes
  - **Cache hit** vs. **cache miss**
  - **Hit ratio**: \# hits / \# memory accesses
What data structure can we use to implement caches?

- Need **FAST** lookups and containment checks
- From CS 240: use a hash table!
- Cache slot = "real address" % CACHE_SIZE

Multiple addresses may map to the same cache slot! (this is called a **conflict**)

Here there are eight slots, so the slot ID is the last 3 bits of the memory address ($2^3 = 8$)
Cache implementations

- A cache line is a block or sequence of bytes that is moved between memory levels in a single operation.
- A cache set is a collection of one or more cache lines.
  - Each cache line contains a tag to identify the source address and a valid flag/bit indicating whether the value is up-to-date.

![Diagram of cache memory organization showing valid bits and tags for different sets and cache lines.](image)
Cache implementations

- General cache organization:
  - $S = \# \text{ of cache sets} = 2^s$
    - $s = \# \text{ of bits for set index}$
  - $E = \# \text{ of lines per cache set}$
    - Level of associativity
  - $B = \text{block (cache line) size} = 2^b$
    - $b = \# \text{ of bits for block offset}$
  - $m = \# \text{ of bits for memory address}$
    - $M = \text{size of memory in bytes} = 2^m$
  - $C = \text{total cache capacity} = S \times E \times B$
  - $t = \# \text{ of tag bits} = m - s - b$

![Cache organization diagram]
Types of caches

- **Direct-mapped** \((E = 1)\)
  - One line per set

- **Set-associative** \((1 < E < C/B)\)
  - Multiple lines per set

- **Fully-associative** \((E = C/B)\)
  - All lines in one set

Here \(E = 2\)
Cache implementations

- **Direct-mapped** \((E = 1)\) caches

![Diagram of direct-mapped cache](image)

- **Set 0**: Valid, Tag, Cache block
- **Set 1**: Valid, Tag, Cache block
- **Set S-1**: Valid, Tag, Cache block

- **Selected set**
  - \(t\) bits
  - \(s\) bits
  - \(b\) bits

- **Tag**: 0 0 0 1

- **Set index**: 0

- **Block offset**: 0

- **Valid bit must be set**

- **Tag bits in the cache line must match the tag bits in the address**

- **If (1) and (2), then cache hit, and block offset selects starting byte.**
Cache implementations

- **Set-associative** \((1 < E < C/B)\) caches

  \[\text{Selected set} \quad \begin{array}{ccc} t \text{ bits} & s \text{ bits} & b \text{ bits} \\ 0 & 0 & 1 \end{array} \]

\[
\begin{array}{|c|c|c|}
\hline
\text{Valid} & \text{Tag} & \text{Cache block} \\
\hline
\text{Valid} & \text{Tag} & \text{Cache block} \\
\hline
\text{Valid} & \text{Tag} & \text{Cache block} \\
\hline
\vdots & \text{Valid} & \text{Tag} & \text{Cache block} \\
\hline
\end{array}
\]

\(E=2\) lines per set

"Two-way set associative"

(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 implementations

- **Fully-associative** \((E = C/B)\) caches

The entire cache is one set, so by default set 0 is always selected.

Set 0:

\[
\begin{array}{ccc}
\text{Valid} & \text{Tag} & \text{Cache block} \\
\text{Valid} & \text{Tag} & \text{Cache block} \\
\vdots & & \\
\text{Valid} & \text{Tag} & \text{Cache block} \\
\end{array}
\]

\(E = C/B\) lines in the one and only set.

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 implementations

- In general, we use the middle bits for the set index
  - Contiguous memory blocks should map to different cache sets
Cache misses (“Three C’s”)

• Compulsory / cold miss
  - First cache miss due to an “empty” cache
  - As the cache loads data, it is warmed up

• Conflict miss
  - Cache miss due to multiple lines in working set mapping to the same cache line
  - Repeated conflict misses for the same cache lines or blocks is called thrashing

• Capacity miss
  - The working set (amount of memory accessed in a given time interval) is too large to fit in cache
Cache policies

- If a cache set is full, a cache miss in that set requires lines to be replaced or evicted.

- Policies:
  - Random replacement
  - Least recently used
  - Least frequently used

- These policies require additional overhead:
  - More important for lower levels of the memory hierarchy.
Cache policies

• How should we handle writes to a cached value?
  – **Write-through**: immediately update to lower level
    • Typically used for higher levels of memory hierarchy
  – **Write-back**: defer update until replacement/eviction
    • Typically used for lower levels of memory hierarchy

• How should we handle write misses?
  – **Write-allocate**: load then update
    • Typically used for write-back caches
  – **No-write-allocate**: update without loading
    • Typically used for write-through caches
Performance impact

• Metrics
  - Hit rate/ratio: \# hits / \# memory accesses (1 – miss rate)
    • Hit time: delay in accessing data for a cache hit
  - Miss rate/ratio: \# misses / \# memory accesses
    • Miss penalty: delay in loading data for a cache miss
  - Read throughput (or "bandwidth"): the rate that a program reads data from a memory system

• General observations:
  - Larger cache = higher hit rate but higher hit time
  - Lower miss rates = higher read throughput
Cache system design involves tradeoffs

- Larger caches => higher hit rate but higher hit time
  - Size vs. speed
- Larger blocks => higher hit rate for programs with good spatial locality, but lower hit rate for others
  - Favor spatial vs. temporal locality
- Higher associativity => lower chance of thrashing but expensive to implement w/ possibly increased hit time
  - Hit time vs. miss penalty
- More writes => simpler to implement but lower performance
  - Write-through vs. write-back
Cache architecture

- Example: Intel Core i7
- Per-core:
  - Registers
  - L1 d-cache and i-cache
    - Data and instructions
  - L2 unified cache
- Shared:
  - L3 unified cache
  - Main memory
Temporal locality

- Working set size vs. throughput
Spatial locality

- Stride vs. throughput
Memory mountain (CS:APP)

- Stride and WSS vs. read throughput

Core i7 Haswell
- 2.1 GHz
- 32 KB L1 d-cache
- 256 KB L2 cache
- 8 MB L3 cache
- 64 B block size

Higher is better
Output of `lscpu`:

- **Architecture**: x86_64
- **Byte Order**: Little Endian
- **CPU(s)**: 24
- **Thread(s) per core**: 2
- **Core(s) per socket**: 6
- **Socket(s)**: 2
- **Vendor ID**: Intel
- **Model name**: Intel(R) Xeon(R) CPU E5-2640
- **CPU max MHz**: 3000.0000
- **CPU min MHz**: 1200.0000
- **L1d cache**: 32K
- **L1i cache**: 32K
- **L2 cache**: 256K
- **L3 cache**: 15360K
Output of `lscpu`:

Architecture: x86_64
Byte Order: Little Endian
CPU(s): 48
Thread(s) per core: 2
Core(s) per socket: 12
Socket(s): 2
Vendor ID: Intel
Model name: Intel(R) Xeon(R) CPU E5-2680
CPU max MHz: 3300.0000
CPU min MHz: 1200.0000
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 30720K
Case study: matrix multiply

(a) Version $ijk$

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += A[i][k]*B[k][j];
        C[i][j] += sum;
    }
```

(b) Version $jik$

```c
for (j = 0; j < n; j++)
    for (i = 0; i < n; i++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += A[i][k]*B[k][j];
        C[i][j] += sum;
    }
```

(c) Version $jki$

```c
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;
    }
```

(d) Version $kji$

```c
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;
    }
```

(e) Version $kij$

```c
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++) {
        r = A[i][k];
        for (j = 0; j < n; j++)
            C[i][j] += r*B[k][j];
    }
```

(f) Version $ikj$

```c
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];
    }
```

Figure 6.44 Six versions of matrix multiply. Each version is uniquely identified by the ordering of its loops.
Case study: matrix multiply

Lower is better
Optimization strategies

- Focus on the common cases
- Focus on the code regions that dominate runtime
- Focus on inner loops and minimize cache misses
- Favor repeated local accesses (temporal locality)
- Favor stride-1 access patterns (spatial locality)

**Tip:** You can use Valgrind to detect cache misses (look up a tool called *cachegrind*)
Next time

- **Virtual memory**: an OS-level memory cache
  - Bridge between module 4 (machine architectures) and module 5 (operating systems)
  - Module 4 unit test over the weekend (due Monday)