Notes and Updates

- Office hours for Dr. Simon: no more this week
- Today:
  - Introduction to Caches
- Thursday:
  - Guest lectures: Cray MTA, Multicore procs
  - THIS WILL BE ON THE FINAL
- Questions?
  - How did last lecture go?

Evaluations: The real deal
Before Quiz on Thursday

- These are very important to me
  - Effort in helping you learn/Activities
  - Technology
  - Fairness/Responsiveness to you
  - Helping you professionally

- CAPE evals versus informal evals

- Questions about “instructor” versus “class”

- Please come and give us your opinion. Take some time to think about this class, my efforts to help you learn, and what you would like to say
Improving memory with caches

The five components
We've seen memory in IF and MEM

- We had memory that could be accessed in 1 cycle
  - instruction memory
  - data memory
- Split between inst and data memory was weird
  - needed to allow pipelining
  - but code can be self-modifying
  - or a compiler's "data" is the "code" of another program

Life without cache (with only main memory)

- CPU
  - 1GHz
- Main memory
  - 30ns access time
- Disk memory
The problem with memory

- It’s expensive (and perhaps impossible) to build a large, fast memory
  - "fast" meaning "low latency"
  - why is low latency important?
- To access data quickly:
  - it must be physically close
  - there can’t be too many layers of logic
- Solution: Move data you are about to access to a nearby, smaller, memory cache
  - Assuming you can make good guesses about what you will access soon.

A typical memory hierarchy

- CPU
- SRAM memory (small, fast)
- DRAM memory (big, slower, cheaper/bit)
- Disk memory (huge, very slow, very cheap)

- on-chip "level 1" cache
- off-chip "level 2" cache
- main memory
- disk
Memory and your program

int data[8] = {1,2,3,4,5,6,7,8}
for (int i = 0; i < 8; i++)
{
    blah += data[i];
    //fprintf(stderr, "%i\n", &data[i]);
}

Assumptions

• Our main memory can hold 8 words
  - 1 word is one data element (an integer)
    • 32 bytes in one word
    • Each data element starts on an address that is a multiple of 4
  - Our data will be at addresses:
    0, 4, 8, 12, 16, 20, 24, 28

• Cache which can hold 4 words (data elements)
  - We’ll look at a few different “designs” of cache
**S1: avg 4 elems, print 4 elems**

```c
int data[8] = {1,2,3,4,5,6,7,8}
Memory Access Pattern:
data[0]
data[1]
data[2]
data[3]
data[0]
data[1]
data[2]
data[3]
```

**S2: avg of even elements**

```c
int data[8] = {1,2,3,4,5,6,7,8}
Memory Access Pattern:
data[0]
data[2]
data[4]
data[6]
data[0]
data[0]
data[2]
data[4]
data[6]
```

NEW CACHE DESIGN:
- white reserved for “evens”
- blue reserved for “odds”
- HINT: use 3rd to last bit to determine even or odd
S1 again: avg of 4 elems, print 4 elems

```c
int data[8] = {1,2,3,4,5,6,7,8};
```

**Memory Access Pattern:**
- `data[0]`
- `data[1]`
- `data[2]`
- `data[3]`
- `data[0]`
- `data[1]`
- `data[2]`
- `data[3]`

NEW CACHE DESIGN:
- 2-line cache each with 2 data elements
- White reserved for “evens”
- Blue reserved for “odds”
- Everytime you are called bring a friend with you (the one who shares your 4th bit)

S3 again: print, change value of 4 elems

```c
int data[8] = {1,2,3,4,5,6,7,8};
```

**Memory Access Pattern:**
- `ld data[4]`
- `st data[4]`
- `ld data[5]`
- `st data[5]`
- `ld data[6]`
- `st data[6]`
- `ls data[7]`
- `st data[7]`

ORIGINAL CACHE DESIGN:
- Each element can go in any location...
Issues we touched on

- How data moves from memory to cache
  - Benefits: Temporal locality
- What to do when cache is full
  - Replacement policies
- Placement options for where data can “go” in cache
  - Direct-mapped, Set-associative, Fully-associative
- Moving “lines”/“blocks” into cache
  - Benefit: Spatial locality
- Writing values in a code
  - Cache/MM out of synch with registers
  - Write back policies in caches

Cache basics

- In running program, main memory is data’s “home location”.
  - Addresses refer to location in main memory.
  - “Virtual memory” allows disk to extend DRAM
    - Address more memory than you actually have (more later)
- When data is accessed, it is automatically moved up through levels of cache to processor
  - “lw” uses cache’s copy
  - Data in main memory may (temporarily) get out-of-date
    - How?
    - But hardware must keep everything consistent.
  - Unlike registers, cache is not part of ISA
    - Different models can have totally different cache design
A typical memory hierarchy

lw $r3, 100($r6)

The principle of locality

Memory hierarchies take advantage of memory locality.
- The principle that future memory accesses are near past accesses.

Two types of locality:
- _______ locality - near in time: we will often access the same data again very soon
- _______ locality - near in space/distance: our next access is often very close to recent accesses.

This sequence of addresses has both types of locality
0, 4, 8, 0, 4, 8, 32, 32, 256, 36, 40, 32, 32...
How does HW decide what to cache?

Taking advantage of temporal locality:
- bring data into cache whenever its referenced
- kick out something that hasn’t been used recently

Taking advantage of spatial locality:
- bring in a block of contiguous data (cacheline), not just the requested data.

Some processors have instructions that let software influence cache:
- Prefetch instruction ("bring location x into cache")
- "Never cache x" or "keep x in cache" instructions

Cache Vocabulary

- **cache hit**: an access where data is already in cache
- **cache miss**: an access where data isn’t in cache
- **Hit time**: time to access the cache
- **miss penalty**: time to move data from further level to closer, then to cpu
- **hit rate**: percentage of time the data is found in the cache
- **miss rate**: (1 - hit rate)
Cache Vocabulary

- **cache block size** or **cache line size**: the amount of data that gets transferred on a cache miss.
- **instruction cache (I-cache)**: cache that can only hold instructions.
- **data cache (D-cache)**: cache that can only hold data.
- **unified cache**: cache that holds both data & instructions.

A typical processor today has separate "Level 1" I- and D-caches on the same chip as the processor (and possibly a larger, unified "L2" on-chip cache), and larger L2 (or L3) unified cache on a separate chip.

Cache Issues

On a memory access -
- How does hardware know if it 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 is where?
A simple cache

<table>
<thead>
<tr>
<th>Address Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 00000100</td>
</tr>
<tr>
<td>8 00001000</td>
</tr>
<tr>
<td>12 00001100</td>
</tr>
<tr>
<td>4 00001000</td>
</tr>
<tr>
<td>8 00001000</td>
</tr>
<tr>
<td>20 00010100</td>
</tr>
<tr>
<td>4 00000100</td>
</tr>
<tr>
<td>8 00001000</td>
</tr>
<tr>
<td>20 00010100</td>
</tr>
<tr>
<td>24 00011000</td>
</tr>
<tr>
<td>12 00001100</td>
</tr>
<tr>
<td>8 00001000</td>
</tr>
<tr>
<td>4 00000100</td>
</tr>
</tbody>
</table>

- A cache that can put a line of data anywhere is called
- The most popular replacement strategy is

A simpler cache

<table>
<thead>
<tr>
<th>Address Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 00000100</td>
</tr>
<tr>
<td>8 00001000</td>
</tr>
<tr>
<td>12 00001100</td>
</tr>
<tr>
<td>4 00001000</td>
</tr>
<tr>
<td>8 00001000</td>
</tr>
<tr>
<td>20 00010100</td>
</tr>
<tr>
<td>4 00000100</td>
</tr>
<tr>
<td>8 00001000</td>
</tr>
<tr>
<td>20 00010100</td>
</tr>
<tr>
<td>24 00011000</td>
</tr>
<tr>
<td>12 00001100</td>
</tr>
<tr>
<td>8 00001000</td>
</tr>
<tr>
<td>4 00000100</td>
</tr>
</tbody>
</table>

- A cache that can put a line of data in exactly one place is called
- What’s the tag in this case?
**Direct-mapped cache**

- Keeping track of when cache entries were last used (for LRU replacement) in big cache needs lots of hardware and can be slow.
- In a direct mapped cache, each memory location is assigned a single location in cache.
  - Usually* done by using a few bits of the address

---

*Some machines use a pseudo-random hash of the address
But it is DETERMINISTIC!*

---

**Or another way to look at it: an 8 entry cache**
**Compare and Contrast Direct Mapped and Fully Associative Caches**

- What were the “hit rates” for each cache design in our example?
- Which has better temporal locality in our example?

---

**Choosing bits for the index**

If line length is $n$ Bytes, the low-order $\log_2 n$ bits of a Byte-address give the $\_\_\_\_\_\_\_\_\_\_$ of address within a line.

The next group of bits is the $\_\_\_\_\_\_\_\_\_$ -- this ensures that if the cache holds $X$ bytes, then any block of $X$ contiguous Byte addresses can co-reside in the cache. (Provided the block starts on a cache line boundary.)

The remaining bits are the $\_\_\_\_\_\_\_\_\_\_\_$

Anatomy of an address:

```plaintext
\_\_\_\_\_\_\_\_\_\_
```
**Direct Mapped Caches vs. Fully Associative Caches**

- Direct mapped was "easy" to implement
  - Each address can go in exactly one "spot" (line) in cache
  - But sometimes addresses used "together" (4,20,4,20) mapped to the same cache location, causing

  ![Address Map](image)

  Address 4 and 20 both map to same line in a 4-entry cache

- Fully Associative always kicked out [least recently used address](#)

---

**Compromise: A set-associative cache**

<table>
<thead>
<tr>
<th>address string:</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

- A cache that can put a line of data in exactly \( n \) places on a line is:
- The cache lines that share the same index are a
A Better Cache Design

- Direct mapped caches are simpler
  - Less hardware; possibly faster
- Fully associative caches usually have fewer misses.
- Set associative caches try to get best of both.
  - An index is computed from the address
  - In a "k-way set associative cache", the index specifies a set of k cache locations where the data can be kept.
    - k=1 is direct mapped.
    - k=cache size (in lines) is fully associative.
  - Use LRU replacement (or something else) within the set.

### 4-way set associative cache

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
<th>tag</th>
<th>data</th>
<th>tag</th>
<th>data</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Effects of Cache Associativity

![Graph showing the effects of cache associativity with different cache sizes](image)

- Different lines show different cache sizes

<table>
<thead>
<tr>
<th>Miss rate</th>
<th>One-way</th>
<th>Two-way</th>
<th>Four-way</th>
<th>Eight-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB</td>
<td>16 KB</td>
<td>32 KB</td>
<td>64 KB</td>
<td>128 KB</td>
</tr>
<tr>
<td>2 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>