354 Today

Class: Part 3(!)

→ Each week:
  some large topic
  in one day

→ Today: Hardware
       Hardware; Caches
// array.c
int data[DIM][DIM];
int main(int argc, char* argv[]) {
    int i, j;
    // ... some init stuff ...
    double start = gettime();
    for (i = 0; i < DIM; i++)
        for (j = 0; j < DIM; j++)
            data[i][j] = random() % 256;
    double end1 = gettime();
    for (j = 0; j < DIM; j++)
        for (i = 0; i < DIM; i++)
            data[i][j] = random() % 256;
    double end2 = gettime();
    // ... some printing stuff ...
}

// sort.c
int compare(const void *a, const void *b) {
    return *(int *)a - *(int *)b;
}

int main() {
    // ... some init stuff ...
    double start = gettime();
    #ifdef POINTER_SORT
    printf("pointer sort\n");
    ptr_t* just_keys = malloc(array_size * sizeof(ptr_t));
    for (i = 0; i < array_size; i++) {
        just_keys[i].key = data[i].key;
        just_keys[i].record_ptr = &data[i];
    }
    start = gettime();
    qsort(just_keys, array_size, sizeof(ptr_t), compare);
    #else
    printf("record sort\n");
    qsort(data, array_size, sizeof(rec_t), compare);
    #endif
    double end = gettime();
    // ... some printing stuff ...
}
record sort

100 bytes

rec-t

rec-t

key

values

key

1

24

sort:
comparisons
+ swap (if need be)
typedef
struct ptr {
  int key;
  rect * record_ptr;
} ptr_t;

ptr_t & key

8 bytes

key

data[0]
data[1]

compare + swap

why is swap better?
less data movement
\( O(n) \) steps

"Complexity" of algorithm

\[
\text{for } i = 0; i < N; i++
\]

\[ n \in N \Rightarrow O(n) \]

\[ \Rightarrow \text{# of steps is not enough to predict performance} \]
Memory:

- CPU
- Memory
  - inst
  - heap
  - stack

Memory is slow

Solutions?

⇒ (registers) ⇒ reg

Problems remain:

⇒ instruction fetch
⇒ accessing a lot of data (more than fits into registers)
Hardware cache:

CPU

registers

cache

main memory

Another memory

processing:

Cache: goal
keep copies of frequently used "data" here

Assumption:
Cache is smaller than main memory but faster

=> large memory => slow
fast memory => small
Goal: How do caches work?

Memory:

- byte address $\rightarrow$ Memory [load]
- data $\rightarrow$ address, data $\rightarrow$ Memory [store]

x86:

movl 10000, %eax

$\Rightarrow$ read 4 bytes from mem
put it into eax (register)

explicit data movement/copy

cache: implicit (hidden)

$\Rightarrow$ make large memory
seem fast
```
CPU:
- fetch
- decode: ah, this is a [load]
- execute:
  - check if address = 1000
  - is in cache
  - if it is: cache hit
    - use cached copy to complete instruction
  - if not in cache: miss
    - fetch data from memory,
    - put copy in cache
    - finish instruction
```
unit/size of access?

movl = 1000, %eax

4 byte

memory: more efficient
if move more data
(larger than 4 bytes)
cache
block/line:
typical size: \( \left[ 32 \text{ bytes}, 64 \text{ bytes}, 128 \text{ bytes} \right] \)
cache blocks:

benefits:

- miss
  - hit
  - hit
  - memory

movl 1024, %eax
movl 1028, %ebx
movl 1032, %ecx

cache block: 32 bytes

\Rightarrow \text{prefetch}
Cache organization

Cache block: 32 bytes

Cache entries: typical size
pretend small
Cache w/ 4 entries
(1000 entries
Level-1 cache)

Problem: given address,
determine hit/miss

Address:
8-bit address
00000000

Address: 255 = 11111111
7 31 offset
Cache: 2 entries

Single entry:

_data (block)_

32-bytes

block address

256 bytes

CPU:
- examine cache, figure out: hit, miss?
- address: `movl 100(%eax), %ebx` 
- `100 + [eax]` 

→ use tag to determine hit/miss
movl 30, %eax  
[unaligned access]

256 bytes

32 bytes

32 bytes

32 bytes

4-entry cache

valid tags

blocks

which block is cached?

1) what happens when system first starts executing?

valid bit: one per entry
2) What happens when cache fills up?

Access: to block \(\text{llll (7)}\)

\(\Rightarrow\) miss + cache full

\(\Rightarrow\) evict / replace existing entry

[eviction/policy replacement]

heuristic?
eviction/replacement policy:

=> furthest away (address) distance

=> timing: kick out block least recently used (LRU)

=> least frequently used (LFU)

=> pseudo-random

4-entry cache:

loop:

1 = 5 different blocks

2

3

4
Problem w/ cache: hard to build efficiently

why? caches have 1000's of entries

⇒ 1000's of tags

Cache lookup:

⇒ has to look thru all tags to see hit/miss

How to build a cache that doesn't have to check all tags at once? (faster)
Fully-associative cache

- block from memory can go anywhere in cache
- slow / complex

Other extreme: Direct-mapped cache

- block, based on address, can only go into one particular slot of cache

Example:

Address

- use some of upper bits to determine slot of cache

```
00 01 10 11
```

- (5124-byte memory) 32-byte block
only look in one place in cache

3 bits

address

valid tag

tag

32-byte block

index

which slot of cache?

movb 31, %al

address: 0000001111111111 = 31 (dec)
char a[size];

for (i=0; i<size; i++)
    a[i] = 0;  movb $0, 0

addresses

<table>
<thead>
<tr>
<th>decimal</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>31</td>
<td>11111111</td>
</tr>
<tr>
<td>32</td>
<td>00000000</td>
</tr>
</tbody>
</table>

cache: block (32 bytes)

- tag: 000
- 000
- 000

1 byte

block: 0

32 byte

a(0) -> miss
a(2) -> hit
sum: %ecx

movl 1024, %eax  \_memory

movl 2048, %ebx  \_math

(addl %ebx, %ecx)

(addl %eax, %ecx)

CPU:

address:

hit/miss

miss

\[ \begin{array}{c}
\text{XXXXXXX} \\
\text{XX} \\
\text{XX} \\
\text{XX} \\
\text{XX} \\
\text{XX} \\
\text{XX} \\
\end{array} \]

c \rightarrow 000
Problems w/ direct mapped cache:

- conflicts

misses caused by limited mapping

```c
int a[size];
int b[size];
for (i = 0; i < size; i++)
    sum += a[i] + b[i];
```

32 byte block

8-entry cache ⇒ 3 bits of cache index
N-way set-associative:

- Have N slots where block can be put in cache

E.g. (2-way set associative)

8 entry cache like (2 & 4-entry caches)
writes: different than reads?

\[ \Rightarrow \text{movl} \%eax, 1024 \]

Store \(\Rightarrow\) addr: 1024 ... 1027

hit?: \(\Rightarrow\) do we update memory?

or just cache?

write back: update cache upon eviction, flush to memory

write through: update cache + through to memory

simpler, more predictable/better perf on read misses
another policy choice

\[\Rightarrow\] write allocate: on miss, bring cache line/bank into memory

\[\Rightarrow\] write no allocate: on miss, just update memory

Everything else about caches in 5 minutes:

\[\Rightarrow\] instructions, data are cached

Split cache CPU
Multiple levels of cache:

1. First level of cache
   - I
   - D
   - 32KB

2. Second level
   - 256KB
   - Unified (both I + D)
   - 3L
   - 8MB

3. Third level
   - bigger, slower

4. Main memory
int a[dim][dim]

for (i=0; i<dim; i++)
    for (j=0; j<dim; j++)
        a[i][j] =

for (j=0; j<dim; j++)
    for (i=0; i<dim; i++)
        a[i][j]