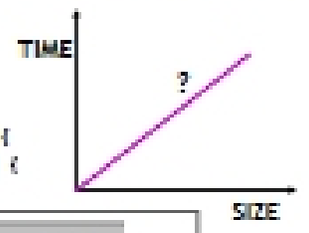
Lecture 14: Caches (I) - Introduction

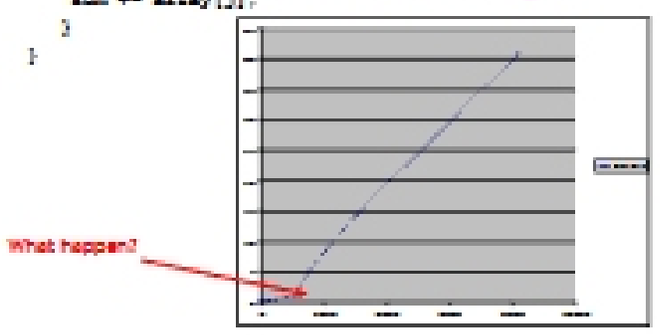
(CPEG323: Intro. to Computer System Engineering)

How will execution time grow with SIZE?

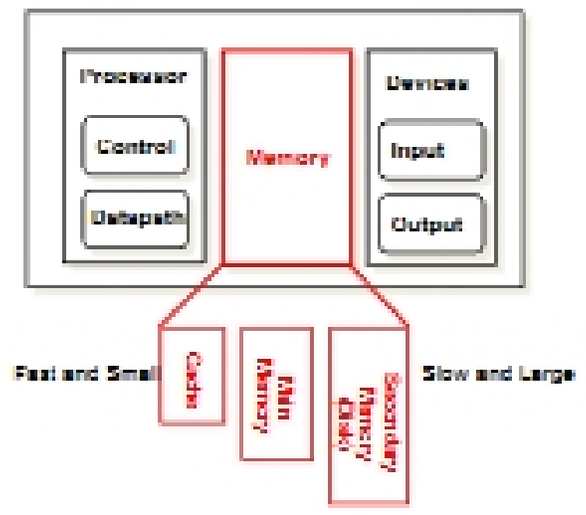
```
int array[SIZE];
int sum = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
   for (int j = 0 ; j < SIZE ; ++ j) {
     sum += array[j];
   }
}
```





Components of a Computer



What we want: Large and fast

- Computers depend upon large and fast storage systems
  - database applications, scientific computations, video, music, etc
  - pipelined CPUs need quick access to memory (IF, MEM)
- So far we've assumed that IF and MEM can happen in 1 cycle
  - unfortunately, there is a tradeoff between speed, cost and capacity

Storage	Delay	Cost/MB	Capacity
Static RAM	1-10 cycles	~$5	128KB-2MB
Dynamic RAM	100-200 cycles	~$0.10	128MB-4GB
Hard disks	10,000,000 cycles	~$0.0005	20GB-400GB

  - fast memory is expensive, but dynamic memory very slow

What we have: Small or slow

- Unfortunately there is a tradeoff between speed, cost and capacity.

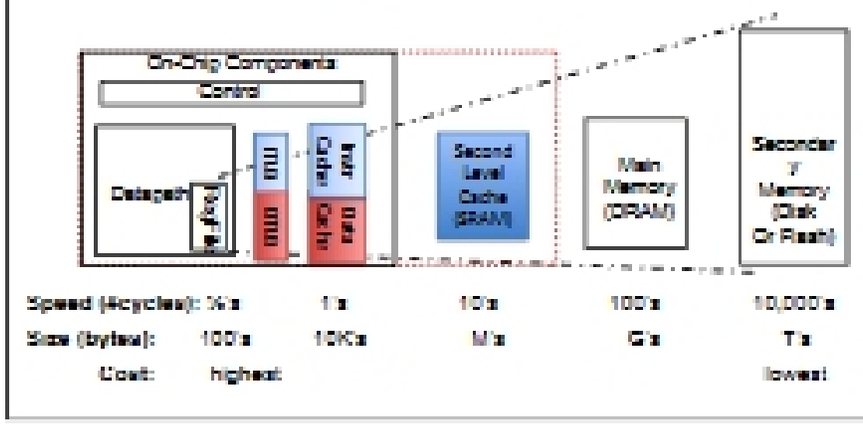
Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

- Fast memory is too expensive for most people to buy a lot of.
- But dynamic memory has a much longer delay than other functional units in a datapath. If every lw or sw accessed dynamic memory, we'd have to either increase the cycle time or stall frequently.
- Here are rough estimates of some current storage parameters.

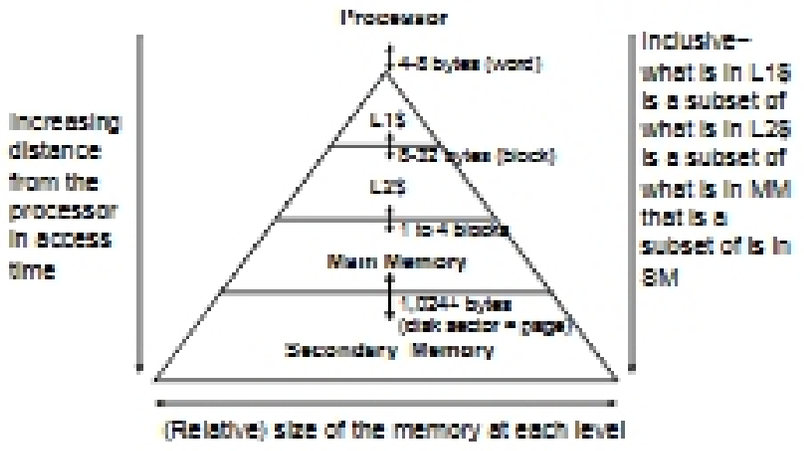
Storage	Delay	Cost/MB	Capacity
Static RAM	1-10 cycles	~$5	128KB-2MB
Dynamic RAM	100-200 cycles	~$0.10	128MB-4GB
Hard disks	10,000,000 cycles	~$0.0005	20GB-400GB

Typical Memory Hierarchy

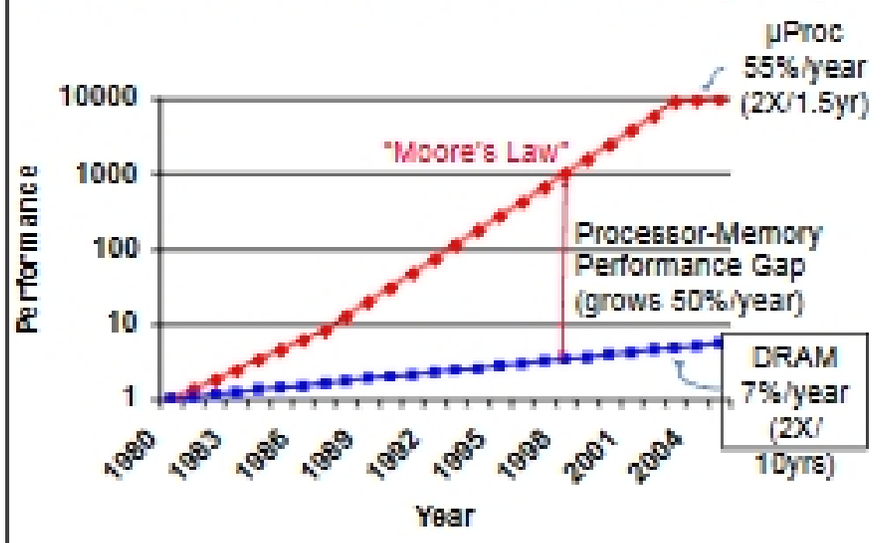
- The Trick: present processor with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology



Characteristics of the Memory Hierarchy

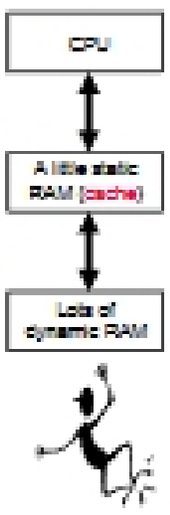


Motivation: Processor-Memory Gap



Introducing caches

- Caches help strike a balance
- A cache is a small amount of fast, expensive memory.
  - It goes between the processor and the slower, dynamic main memory.
  - It keeps a copy of the most frequently used data from the main memory.
- Memory access speed increases overall, because we've made the common case faster.
  - Reads and writes to the most frequently used addresses will be serviced by the cache.
  - We only need to access the slower main memory for less frequently used data.
- Principle used elsewhere: Networks, OS, Search…



The principle of locality

- Usually difficult or impossible to figure out what data will be "most frequently accessed" before a program actually runs
  - hard to know what to store into the small, precious cache memory.
- In practice, most programs exhibit *locality*, which the cache can take advantage of
  - The principle of **temporal locality** says that if a program accesses one memory address, there is a good chance that it will access the same address again.
  - The principle of **spatial locality** says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

Locality in programs

- **Temporal locality:**
  - Loops are excellent examples
    - The loop body will be executed many times.
    - The computer will need to access these same few locations of the instruction memory repeatedly.
  - For example:

```
Loop:  lw    $t0, 0($a1)
       add   $t0, $t0, $a2
       sw    $t0, 0($a1)
       addi  $a1, $a1, -4
       bne   $a1, $0, Loop
```

- **Spatial locality:**
  - Nearly every program, because instructions are usually executed in sequence.

Locality in data

- **Temporal locality**
  - Programs often access the same variables over and over, especially within loops.

```
sum = 0;
for (i = 0; i < MAX; i++)
   sum = sum + f(i);
```

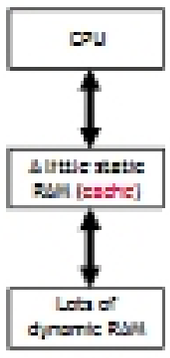
- **Spatial locality**
  - When reading location A from main memory, a copy of that data is placed in the cache but also copy A+1, A+2, …
    - Useful for arrays, records, multiple local variables

```
sum = 0;
for (i = 0; i < MAX; i++)
   sum = sum + a[i];
```

```
employee.name = "Homer Simpson";
employee.boss = "Mr. Burns";
employee.age = 45;
```

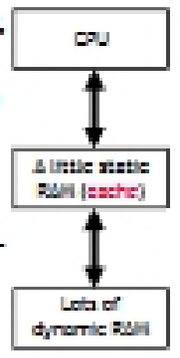
How caches take advantage of temporal locality

- The first time the processor reads from an address in main memory, a copy of that data is also stored in the cache.
  - The next time that same address is read, we can use the copy of the data in the cache *instead* of accessing the slower dynamic memory.
  - So the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster.
- This takes advantage of temporal locality—commonly accessed data is stored in the faster cache memory.



How caches take advantage of spatial locality

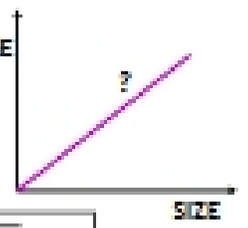
- When the CPU reads location *i* from main memory, a copy of that data is placed in the cache.
- But instead of just copying the contents of location *i*, we can copy *several* values into the cache at once, such as the four bytes from locations *i* through *i* + 3.
  - If the CPU later does need to read from locations *i* + 1, *i* + 2 or *i* + 3, it can access that data from the cache and not the slower main memory.
  - For example, instead of reading just one array element at a time, the cache might actually be loading four array elements at once.
- Again, the initial load incurs a performance penalty, but we're gambling on spatial locality and the chance that the CPU will need the extra data.

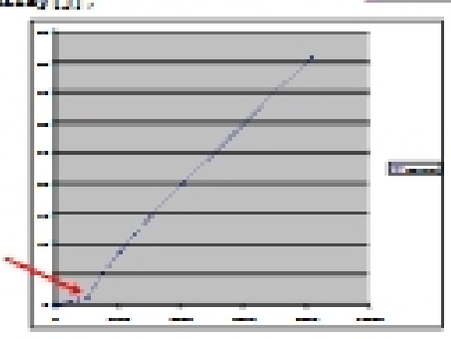


How will execution time grow with SIZE?

```
int array[SIZE];
int sum = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
   for (int j = 0 ; j < SIZE ; ++ j) {
    sum += array[j];
   }
}
```





17

Review so far

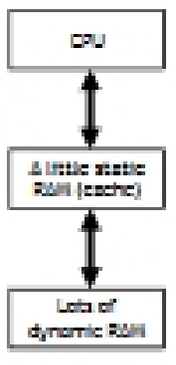
- Wanted: size of the largest memory available speed of the fastest memory available
- Approach: Memory Hierarchy
  - Successively lower levels contain "most used" data from next higher level
  - Exploits *temporal & spatial locality*

Definitions: Hits and misses

- A **cache hit** occurs if the cache contains the data that we're looking for. ☺
- A **cache miss** occurs if the cache does not contain the requested data. ☹
- Two basic measurements of cache performance.
  - the **hit rate** = the percentage of memory accesses that are handled by the cache. (**miss rate** =1 - hit rate)
  - The **miss penalty** = the number of cycles needed to access main memory on a cache miss
- Typical caches have a hit rate of 95% or higher.
- Caches organized in **levels** to reduce miss penalty

Memory System Performance

- Memory system performance depends on three important questions:
  - How long does it take to send data from the cache to the CPU?
  - How long does it take to copy data from memory into the cache?
  - How often do we have to access main memory?
- There are names for all of these variables:
  - The **hit time** is how long it takes data to be sent from the cache to the processor. This is usually fast, on the order of 1-3 clock cycles.
  - The **miss penalty** is the time to copy data from main memory to the cache. This often requires dozens of clock cycles (at least).
  - The **miss rate** is the percentage of misses.



20