CSCI2467: Systems Programming Concepts

Slideset 9: The Memory Hierarchy

Source: CS:APP Chapter 6, Bryant & O'Hallaron

Instructor: Matthew Toups

Spring 2020

THE UNIVERSITY of NEW ORLEANS
DEPARTMENT OF COMPUTER SCIENCE

Shell lab

- do not use `waitpid` in `eval`! **Use `waitfg` in eval.**

Shell lab

- do not use `waitpid` in `eval`! **Use** `waitfg` **in eval.**
- the names `waitfg` and `waitpid` can be confused, but they differ a lot

Shell lab

- do not use `waitpid` in `eval`! **Use `waitfg` in eval.**
- the names `waitfg` and `waitpid` can be confused, but they differ a lot
- You should use `waitpid()` to reap in `sigchld_handler`

Shell lab

- do not use `waitpid` in `eval`! **Use `waitfg` in eval.**
- the names `waitfg` and `waitpid` can be confused, but they differ a lot
- You should use `waitpid()` to reap in `sigchld_handler`
- `waitfg` can be pretty simple and just checks the jobs list

Shell lab

- do not use `waitpid` in `eval`! **Use `waitfg` in eval.**
- the names `waitfg` and `waitpid` can be confused, but they differ a lot
- You should use `waitpid()` to reap in `sigchld_handler`
- `waitfg` can be pretty simple and just checks the jobs list
- You will want to use `fgpid()` and/or `getjobpid()`

Shell lab

- do not use `waitpid` in `eval`! **Use `waitfg` in eval.**
- the names `waitfg` and `waitpid` can be confused, but they differ a lot
- You should use `waitpid()` to reap in `sigchld_handler`
- `waitfg` can be pretty simple and just checks the jobs list
- You will want to use `fgpid()` and/or `getjobpid()`

  putting a while loop around `sleep(1)` is fine

Shell lab and pointers

- Confused about `jobs` and `struct job_t *j`?

  `struct job_t jobs[MAXJOBS]; /* The job list */`

- this is an *array* of `structs`
- `jobs[0]` is a single job
- `jobs` (no index) is a `pointer` to the beginning of the array

  `struct job_t *j; /* pointer to one job struct */`

- see examples in `addjob()` and `clearjob()`

  (helper functions given at the bottom of `tsh.c`)

Shell lab and pointers

```
/* addjob - Add a job to the job list */
int addjob(struct job_t *jobs, pid_t pid, int state,
    char *cmdline)
{
  int i;

  for (i = 0; i < MAXJOBS; i++) {
    if (jobs[i].pid == 0) {
      jobs[i].pid = pid;
      jobs[i].state = state;
      jobs[i].jid = nextjid++;
      if (nextjid > MAXJOBS)
        nextjid = 1;
      strcpy(jobs[i].cmdline, cmdline);
      if(verbose){
        printf("Added job [%d] %d %s\n", jobs[i].jid,
            jobs[i].pid, jobs[i].cmdline);
      }
      return 1;
    }
```

Shell lab and pointers

```
/* clearjob - Clear the entries in a job struct */
void clearjob(struct job_t *job) {
    job->pid = 0;
    job->jid = 0;
    job->state = UNDEF;
    job->cmdline[0] = '\0';
}
```

- In C, if we have:

  `struct my_t s;`

  `struct my_t *sp = &s;`

  `s.a` accesses a as a member of struct s

  `sp->a` dereferences sp, then accesses member a

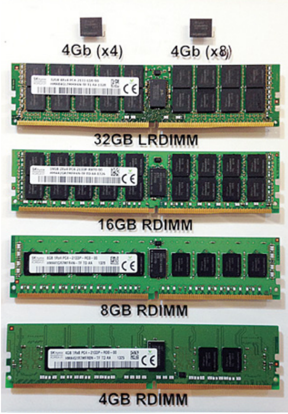
  equivalent to: `(*sp).a`
- See p.131-132 of K&R for more

Today

- Class notes
1. Computer Memory
   - Underlying technologies
   - The crucial challenge in systems design
   - Memory hierarchies
2. Locality
   - Principles and types
   - Examples
3. Caching
   - Caching illustrated
   - General Concepts
   - Cache associativity
   - Cache misses
   - Cache writes
   - Performance

Random-Access Memory (RAM)

- **Key Features**
  - RAM is typically packaged as a chip
  - Basic storage unit is normally a cell (one bit per cell)
  - Multiple RAM chips form modules
- **RAM comes in two varieties:**
  - SRAM (Static RAM)
  - DRAM (Dynamic RAM)



SRAM vs DRAM summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

Trans. $\rightarrow$ transistors

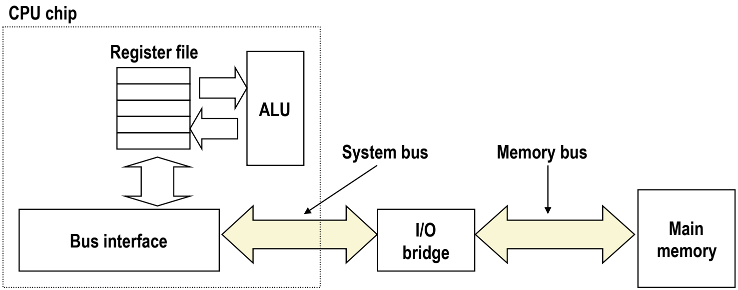
EDC $\rightarrow$ error-detecting code

Nonvolatile memories
Types and uses

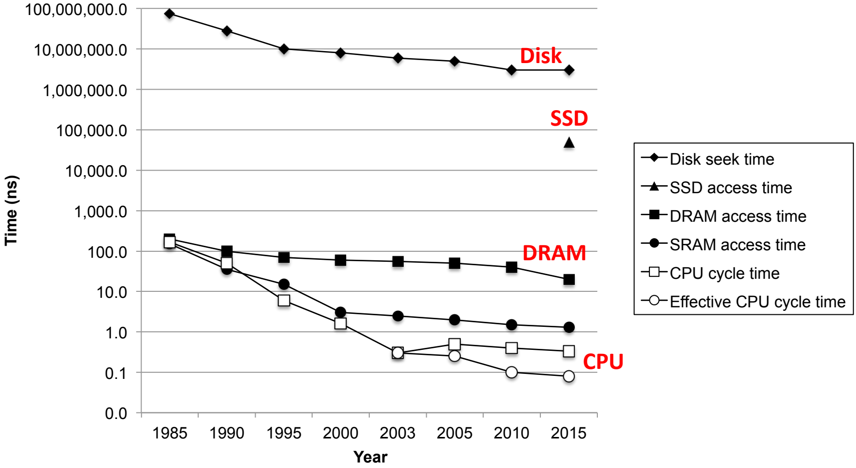
- **DRAM and SRAM are volatile memories**
  - Lose information if powered off.
- **Nonvolatile memories retain value even if powered off**
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically eraseable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs. with partial (block-level) erase capability
    - Wears out after about 100,000 erasings
- **Uses for Nonvolatile Memories**
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,...)
  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,...)
  - Disk caches

Traditional Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



The gap widens between DRAM, disk, and CPU speeds



The CPU-memory gap

This gap has driven systems design for decades

- CPU can execute billions of instructions per second

The CPU-memory gap

This gap has driven systems design for decades

- CPU can execute billions of instructions per second
- CPU gets instructions from main memory (DRAM)

The CPU-memory gap

This gap has driven systems design for decades

- CPU can execute billions of instructions per second
- CPU gets instructions from main memory (DRAM)
- Main memory (DRAM) can service tens or hundreds of millions of accesses per second

The CPU-memory gap

This gap has driven systems design for decades

- CPU can execute billions of instructions per second
- CPU gets instructions from main memory (DRAM)
- Main memory (DRAM) can service tens or hundreds of millions of accesses per second
- Bottleneck! How can we utilize the speed of the CPU?

The CPU-memory gap

This gap has driven systems design for decades

- CPU can execute billions of instructions per second
- CPU gets instructions from main memory (DRAM)
- Main memory (DRAM) can service tens or hundreds of millions of accesses per second
- Bottleneck! How can we utilize the speed of the CPU?
- This is a major dilemmma facing hardware and software designers

Today

- Class notes
1. Computer Memory
   - Underlying technologies
   - The crucial challenge in systems design
   - Memory hierarchies
2. Locality
   - Principles and types
   - Examples
3. Caching
   - Caching illustrated
   - General Concepts
   - Cache associativity
   - Cache misses
   - Cache writes
   - Performance

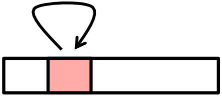
Locality to the rescue!

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality

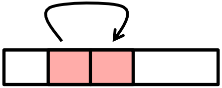
Locality: priciple and types

- **Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time



Locality example

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

- Data references
- reference array elements in succession:
- reference variable `sum` each iteration:
- Instruction references
- reference instructions in sequence:
- cycle through loop repeatedly:

Locality example

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

- Data references
  - reference array elements in succession: Spatial locality
  - reference variable `sum` each iteration:
- Instruction references
  - reference instructions in sequence:
  - cycle through loop repeatedly:

Locality example

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

- Data references
  - reference array elements in succession: Spatial locality
  - reference variable `sum` each iteration: Temporal locality
- Instruction references
  - reference instructions in sequence:
  - cycle through loop repeatedly:

Locality example

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

- Data references
  - reference array elements in succession: Spatial locality
  - reference variable `sum` each iteration: Temporal locality
- Instruction references
  - reference instructions in sequence: Spatial locality
  - cycle through loop repeatedly:

Locality example

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

- Data references
  - reference array elements in succession: Spatial locality
  - reference variable `sum` each iteration: Temporal locality
- Instruction references
  - reference instructions in sequence: Spatial locality
  - cycle through loop repeatedly: Temporal locality

Locality Example

- **Question:** Does this function have good locality with respect to array `a`?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Qualitative Estimates of Locality

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- **Question:** Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

- **Question: Can you permute the loops so that the function scans the 3-d array `a` with a stride-1 reference pattern (and thus has good spatial locality)?**

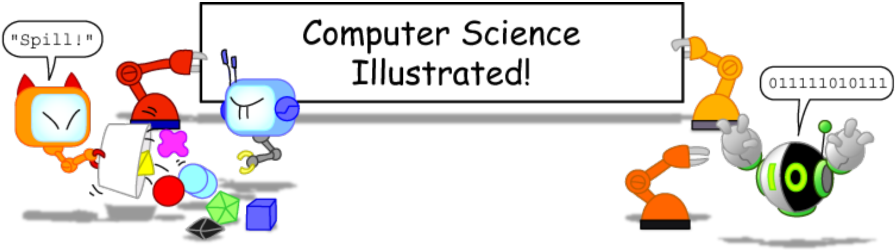
```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

Memory hierarchies

- **Some fundamental and enduring properties of hardware and software:**
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- **These fundamental properties complement each other beautifully.**
- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy.**

Today

- Class notes
1. Computer Memory
   - Underlying technologies
   - The crucial challenge in systems design
2. Locality
   - Principles and types
   - Examples
   - Memory hierarchies
3. Caching
   - Caching illustrated
   - General Concepts
   - Cache associativity
   - Cache misses
   - Cache writes
   - Performance

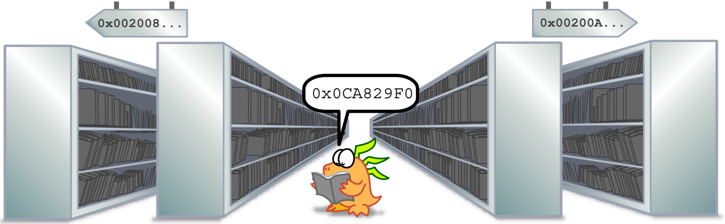


Source: `http://csillustrated.berkeley.edu`

Caches: What Are They For?

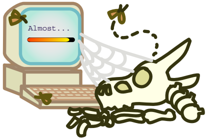
For computers, memory accesses are like going to the library,

Finding the necessary information in the page of a book,



And going back home to do the work involving that information.

While computers don't mind going back and forth like this for data, it usually means users have to do a lot of waiting.



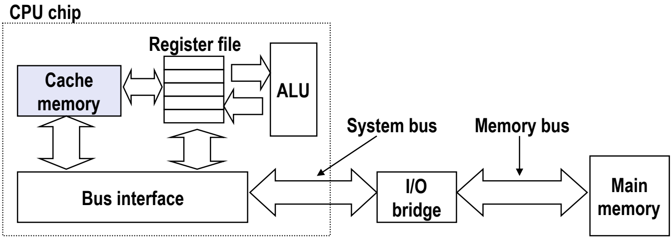
Fortunately for users, computers have caches, which is the equivalent of keeping copies of the books needed on a shelf near the workspace. Through a number of mechanisms, caches give the illusion of being able to access memory very quickly!

Ketrina Yim

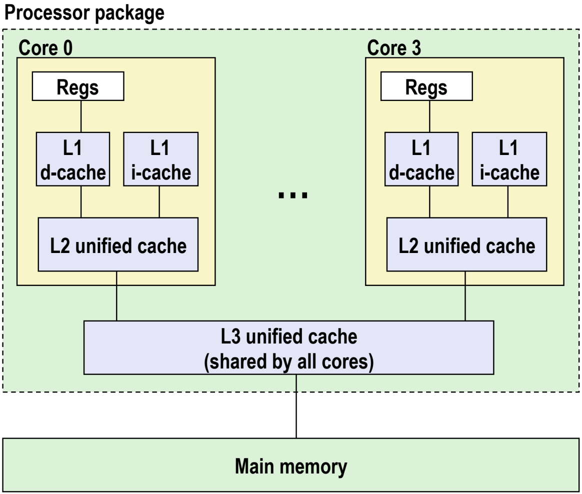
Source: CS Illustrated

- ***Cache:*** **A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.**
- **Fundamental idea of a memory hierarchy:**
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- ***Big Idea:*** **The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.**

- **Cache memories are small, fast SRAM-based memories managed automatically in hardware**
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in cache**
- **Typical system structure:**



Intel Core i7 Cache Hierarchy



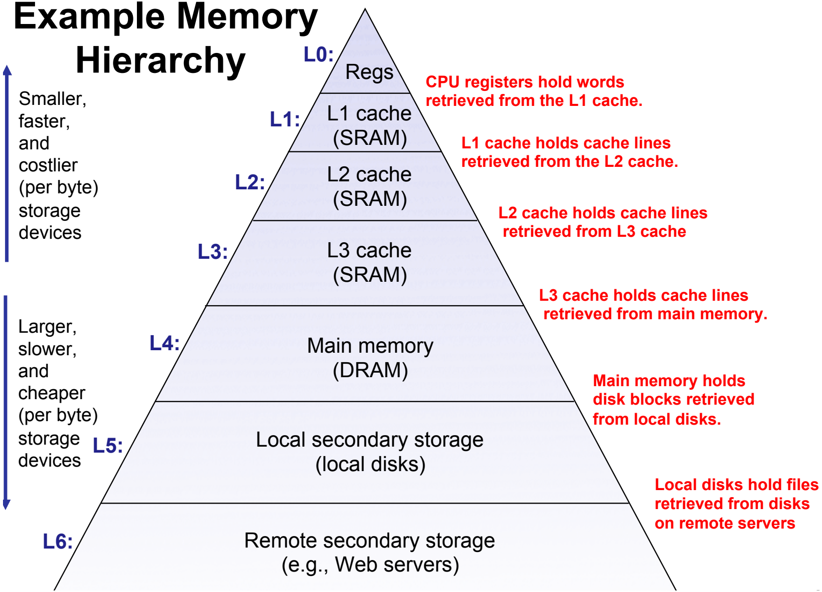
L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

Example Memory Hierarchy

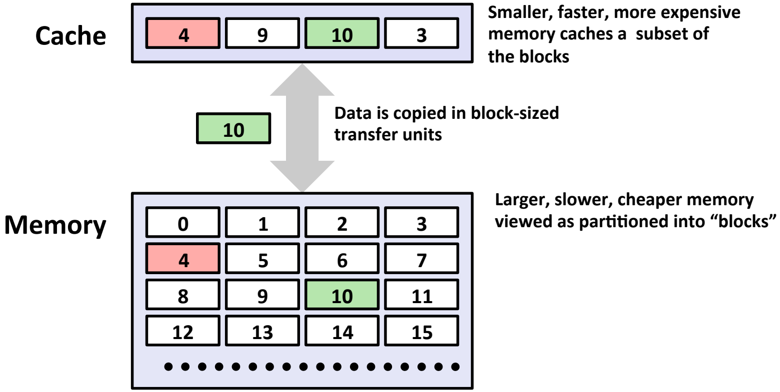


CS:APP3e Figure 6.21

Examples of Caching in the Mem. Hierarchy

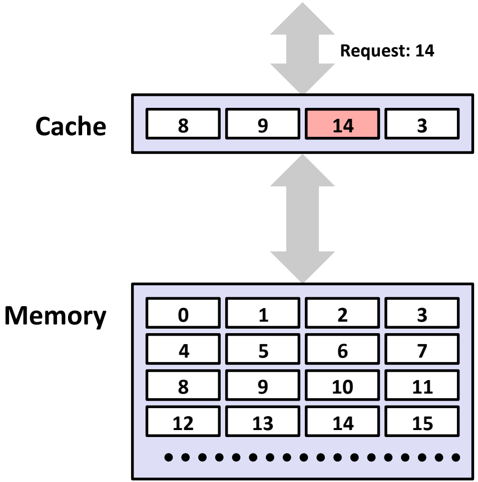
Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

General Cache Concepts



CS:APP3e Figure 6.22

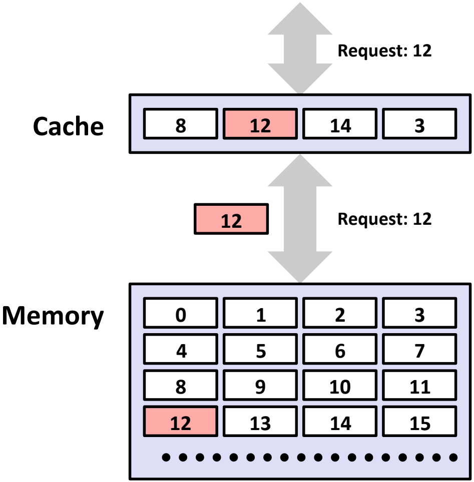
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

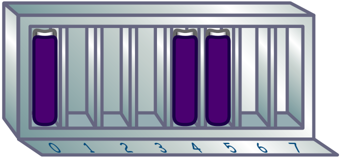
Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Cache Associativity

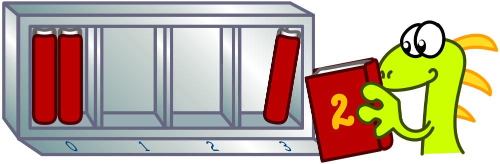
Just as bookshelves come in different shapes and sizes, caches can also take on a variety of forms and capacities. But no matter how large or small they are, caches fall into one of three categories: direct mapped, n-way set associative, and fully associative.

Direct Mapped



A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

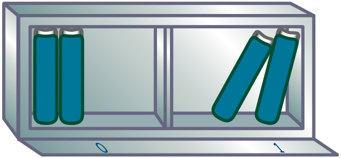
2-Way Set Associative



Tag	Index	Offset

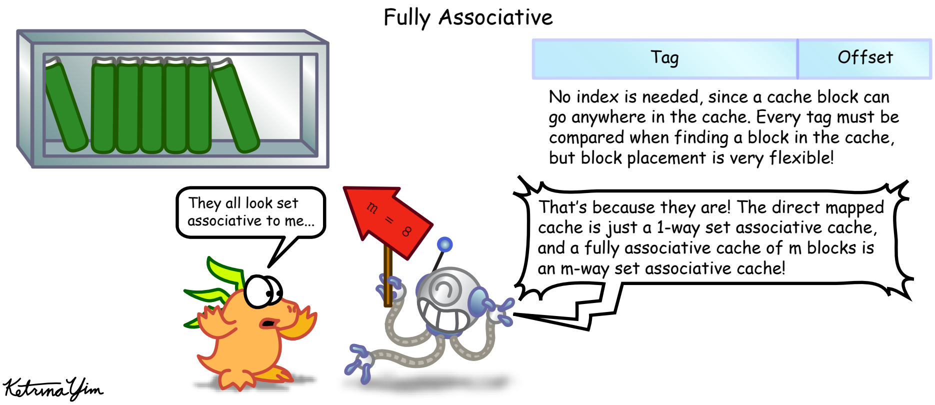
This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

4-Way Set Associative



Tag	Index	Offset

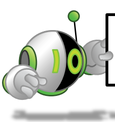
Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.



Source: CS Illustrated

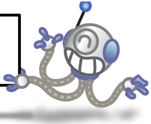
General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.
- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

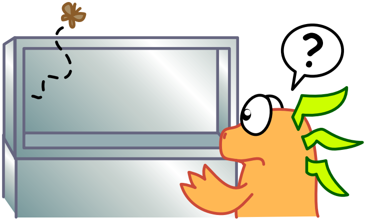


Cache Misses

When you just can't find what you're looking for...



Sometimes, the cache doesn't have the memory block the computer's looking for. When this happens, it's called a cache miss. There are three causes of cache misses. Just remember the three C's:

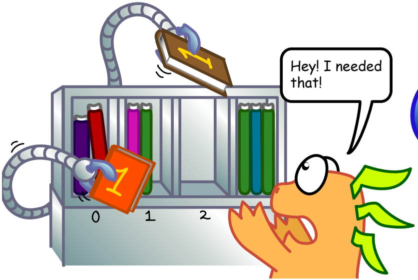


Compulsory

Compulsory misses happen when a block is referenced for the first time. The computer can't get a block that doesn't exist yet!

Capacity

The block is not in the cache because there is no space in the cache for it. Caches are of finite size, after all.



Conflict

These types of misses happen only in direct-mapped and set-associative caches. Multiple blocks can be mapped to a set, forcing evictions when the set is full.

Source: CS Illustrated

What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

Memory hierarchy summary

- **The speed gap between CPU, memory and mass storage continues to widen.**
- **Well-written programs exhibit a property called *locality*.**
- **Memory hierarchies based on *caching* close the gap by exploiting locality.**

Cache performance metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 4 clock cycle for L1
    - 10 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

Consider these numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**

Consider these numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:

    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**

    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

- **This is why "miss rate" is used instead of "hit rate"**

Cache Summary

- **Cache memories can have significant performance impact**

- **You can write your programs to exploit this!**
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.