Digital System Architectures

141IN (last 3 CFU part)

Eric Medvet

A.Y. 2021/2022

Lecturer

Eric Medvet

- Associate Professor of Computer Engineering at Departmenet of Engineering and Architecture, University of Trieste
- Online at: medvet.inginf.units.it

Research interests:

- Evolutionary Computation
- Machine Learning applications
- Embodied intelligence

Labs:

- Evolutionary Robotics and Artificial Life lab
- Machine Learning lab

Materials

Teacher's slides:

- available here
- might be updated during the course

Intended usage:

- slides should contain every concept that has to be taught/learned
- **but**, slides are designed for consumption during a lecture, they might be suboptimal for self-consumption $\Rightarrow$ **take notes!**

Teacher's slides are based on :

- Patterson, David A., and John L. Hennessy. *Computer organization and design ARM edition: the hardware software*

Exam

Written test with questions and short open answers

Course content

In brief:

1. Cache
2. Virtual memory

(See the syllabus!)

Memory: does it matter?

Registers and memory

Processor operates on data through instructions:

- instructions operate on registers
- when needed, **load** (*read*) data from memory to registers
- when needed, **store** (*write*) data from registers to memory

Loads and stores take time!

Overall, how long?

- depends on **number** of operations
- depends on **duration** of single operation

Load/store: how many? (number)

C:

```c
float fahreneit2celsius(float f) {
  return ((5.0 / 9.0 ) * (f - 32.0));
}
```

LEGv8:

```
LDURS S16, [X27, const5]  ; S16 = 5.0 (5.0 in memory)
LDURS S18, [X27, const9]  ; S18 = 9.0 (9.0 in memory)
FDIVS S16, S16, S18       ; S16 = 5.0 / 9.0
LDURS S18, [X27, const32] ; S18 = 32.0
FSUBS S18, S12, S18       ; S18 = f - 32.0
FMULS S0, S16, S18        ; S0 = (5/9)*(fahr - 32.0)
BR LR                     ; return
```

3 loads (LDURS); might be 2 with compiler optimizations

More complex case

```
// 32x32 matrices
void matrixMult (double c[][], double a[][], double b[][]) {
  int i, j, k;
  for (i = 0; i < 32; i = i + 1) {
    for (j = 0; j < 32; j = j + 1) {
      for (k = 0; k < 32; k = k + 1) {
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
    }
  }
}
```

Many more loads and stores!

Load/store: how long? (duration)

Depends on the memory technology:

Type	Access time [ns]	Cost [$/GB]
SRAM	0.5–2.5	500–1000
DRAM	50–70	10–20
Flash	5000–50000	0.75–1.00
Magnetic disk	5000000–20000000	0.05–0.10

Recall: 1 CPU cycle takes $\approx$ 1 ns

(Data from 2012)

(We'll see more later)

Goal

Goal: process more data, faster, cheaper

Ideally, we want to minimize overall time spent accessing memory.

We can work on:

1. number of accessess (depends *mainly* on the **code**)
2. duration of each access

How?

1. write better (optimized) code
2. use faster memory?

→ access time/cost trade-off!

Guy at the library

A guy is doing some research about some topic. Needs to:

- take a book from shelf (long walk)
- read some part
- put back the book on shelf (long walk)

Library ⇿ computer

- Book ⇿ data
- Shelf ⇿ "main" (farther) memory
- Desk ⇿ "local" (closer) memory
- Guy ⇿ processor

Idea: two memories

Suppose to have:

- close memory M_1: small, fast access
- large memory M_2: large, slow access

Whenever you look for data item x:

- if in M_1, take it from there
- otherwise
  - take it from M_2
  - put it in M_1 (freeing some space, if needed)

Useful if you **soon** access again x

(x is the address of the data item)

Better idea

Whenever you look for data item x:

- if in M_1, take it from there
- otherwise
  - take $(\dots, x-2, x-1, x, x+1, x+2, \dots)$ from M_2
  - put *them* in M_1 (freeing *more* some space, if needed)

Useful if you **soon** access again x or something **close** to x!

Locality principle

Useful if you **soon** access again x or something **close** to x!

Temporal locality: if a data location x is accessed at t, it will be likely accessed again **soon** (at some $t + \delta t$)

Spatial locality: if a data location x is accessed at t, data locations **close** to x (some $x + \delta x$) will be likely accessed again soon

[How to exploit spatial locality at the library? What's the assumption?]

Localities in action

More than one level: memory hierarchy

Instead of just two memories M_1 (fast and small) and M_2 (large and slow), a **hierarchy of memories**:

- CPU
- M_1: closest to CPU, fastest, smallest
- M_2: a bit farther, a bit slower, a bit larger
- M_3: a bit farther, a bit slower, a bit larger
- ...
- M_k: far, slow, large

Memory hierarchy

- M_1: closest to CPU, fastest, smallest
- ...
- M_k: far, slow, large

Pros:

- **size** of M_k (the largest)
- (almost) **access time** of M_1 (the closest)
- the overall **cost** is much lower than a M with the size of M_k and the access time of M_1 (ideality)

Cons:

- need to implement the access algorithm **within** the CPU/system

Cache

The memory hierarchy is a *pattern*, a scheme of solution for a common problem

Other cases:

- local copy of network data
- storing of complex computation results
- physical memory

Common name for the closer memory: **cache**

Memory technology

(very briefly)

Recap

Type	Access time [ns]	Cost [$/GB]
SRAM	0.5–2.5	500–1000
DRAM	50–70	10–20
Flash	5000–50000	0.75–1.00
Magnetic disk	5000000–20000000	0.05–0.10

How do they work?

Static Random Access Memory (SRAM)

Properties:

- fixed access (read/write) time
  - read and write times may be different

Technology:

- 6–8 transistors for each bit (for minimizing disturbance during reads): large footpring on silicon
- no need to refresh: access time is very close to cycle time
- low power consumption in standby
- "currently" SRAM are no longer separate chips, but are integrated onto processors

Dynamic Random Access Memory (DRAM)

Properties:

- faster access to batches of data (rows are buffered)

Technology:

- 1 transistor + 1 capacitor per bit: lower footprint than SRAM
- stored data has to be refreshed (read and write)
  - no access while refreshing
  - bits are organized in rows, rows are freshed at once, rows are buffered

"Modern" DRAM

- *syncrhonous DRAM* (SDRAM): a clock facilitates burst transfer
- *Double Data Rate (DDR) SDRAM*: transfer on both rising and falling edge
  - DDR4-3200 means 1600 MHz clock
  - organized in banks: reads/writes done concurrently on several banks
- often physically packed in *dual inline memory modules* (DIMMs)
  - a DIMM usually contains 4-16 DRAM chips
  - a typical DIMM can transfer 25600 MB/s (PC25600)

DRAM over time

Year	Chip size [bit]	Cost [$/GB]	Access time new[1] [ns]	Access time existing[2] [ns]
1980	64 K	1500000	250	150
1983	256 K	500000	185	100
1985	1 M	200000	135	40
1989	4 M	50000	110	40
1992	16 M	15000	90	30
1996	64 M	10000	60	12
1998	128 M	4000	60	10
2000	256 M	1000	55	7
2004	512 M	250	50	5
2007	1 G	50	45	1.25
2010	2 G	30	40	1
2012	4 G	10	35	0.8

Flash memory

Technology:

- based on *electrically erasable programmable read-only memory* (EEPROM)
- writes can wear hardware behind bits
  - a controller takes care of distributing usage evenly

Solid state drives (SSDs) use flash memory

Disk memory

Technology:

- with parts in movements (differently than SRAM, DRAM, flash)

Hard disk drives (HDDs) are storage devices based on disk memory

Terminology

Mechanical parts:

- platters
- heads

Logical parts:

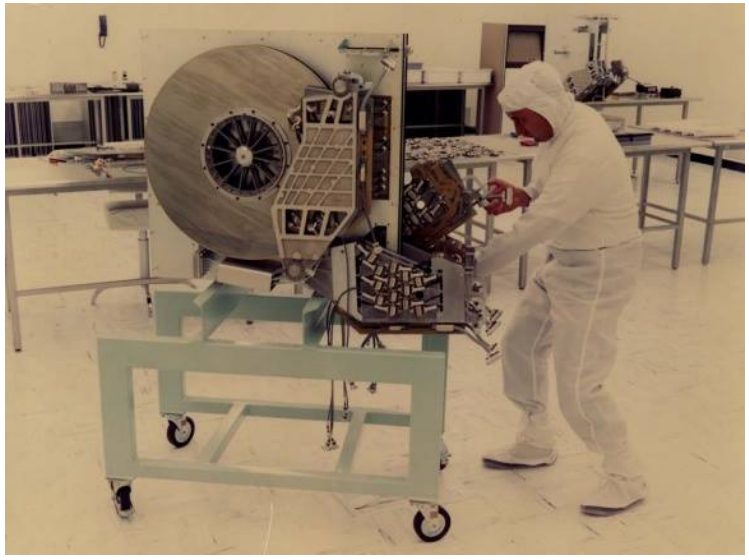
- track (1000s of sectors): concentric cyrcle on the platter
- sector (512–4096 bytes): portion of the track
- cylinder: set of vertically aligned sectors

Read/write phases:

- seek: head moves toward track ($\approx$ 3–13 ms)
- rotational latency or delay ($\approx$ 6 ms @ 5400 RPM)
- actual data read/write

HDD "history"

Different sizes over time



HDD in 1979!

Caches

What's a cache?

Cache: *a hiding place* to put something for *future use*

First usage of the term in computers denoted specifically the small memory between the processor and the main memory

Nowadays, cache denotes any storage that takes advantage of locality

- e.g., browser cache

Using a cache

(Consider just reads for now)

Whenever the processor want to read a memory location x:

- check if x is in the cache (x is the address of the data)
- if yes, read it from there, otherwise, read it from main memory and store it in the cache

How to?

1. know if x is in the cache?
2. if yes, know where it is in the cache? (x is the address of the location in the main memory, not in the cache; if it was, the size of the cache would be the same of the main memory → useless!)

Premise

Memory content vs. memory addresses

```
000 13
001 45
010 e4
011 14
100 0f
101 76
110 1a
111 ff
```

8 byte memory:

- addresses goes from 000_2 = 0 to 111_2 = 7, i.e., 3-bit addresses
- at each address, there is a byte
  - $13_{16} = 00010011_2$
- $[x]$ is the data at x
  - $[011] = 14_{16} = 00010100_2$

In general, there are n-bit addresses (e.g., $n = 64$)

Direct mapped cache

The cache consists of $s_c = 2^{n_c}$ triplet $\langle$**validity bit**, **tag**, **block**$\rangle$

- a **block** contains $s_b = 2^{n_b}$ bytes (e.g., $s_b = 8, 64, \ldots$)
  - the k-th block will host portions of the main memory of s_b bytes starting at addresses x s.t. $x \,\%\, s_c = k$
  - $\Rightarrow$ many different x map to the same block
- a **tag** indicates which one of the many different x is actually in the block
  - there can be $2^{n_m - n_c - n_b}$ values, with main memory size $s_m = 2^{n_m}$
- a **validity bit** indicates if the block is to be considered valid
  - initially set to `0`

Example

Looking in the cache

(Assume $s_b = 1$)

Given a cache with $s_c = 2^{n_c}$ blocks, looking for x means looking at the k-th block with $x \,\%\, s_c = k$

With binary numbers $x \,\%\, s_c = x \,\%\, 2^{n_c}$ is "the less significant n_c bits of x"

Example with $n_m = 8, n_c = 3$:

- $x = 01001\underline{111}_2 = 79 \Rightarrow k = 79 \,\%\, 2^3 = 79 \,\%\, 8 = 7 = 111_2$
- $x = 00101\underline{011}_2 = 43 \Rightarrow k = 43 \,\%\, 2^3 = 43 \,\%\, 8 = 3 = 011_2$

One to many

Cache

($s_c = 4, n_c = 2, s_b = 1$)

```
00 1 01 00010000
01 1 10 10000010
10 0 11 10000010
11 0 11 10010101
```

Main memory

($n_m = 4, s_b =$ 16 bytes = 128 bit)

```
0000 00000001
0001 00000010
0010 00000100
0011 00001000
0100 00010000
0101 00100000
0110 01000000
0111 10000000
1000 10000001
1001 10000010
1010 10000100
1011 10001000
1100 10010000
1101 10100000
1110 11000000
1111 11000001
```

Cache size

1. For a main memory of 4 GB and a cache with 1024 blocks of 4 words each
   - compute the tag size in bits
   - compute the ratio between actual data stored in the cache and overall cache size
2. Repeat for a main memory of 4 GB and a cache with 1024 blocks of 16 words each

(1 word is 4 bytes)

Algorithm for reading x

Given x of n_m bits (**assume** $s_b = 1$):

1. take $y = x_{[n_m - n_c, n_m[}$ as the n_c less significant bits of x
2. read triplet $t = [y]$ from cache at address y
   - $t_{\text{val}} = t_{[0,1[}$ is the first bit of t
   - $t_{\text{tag}} = t_{[1,1+(n_m-n_c)[}$ are bits of t from 2-nd to $2 + (n_m - n_c)$-th
   - $t_{\text{block}} = t_{[1+(n_m-n_c),1+(n_m-n_c)+8[}$ is the block (of 1 byte)
3. if $t_{\text{val}} \neq 1$, go to 6
4. if $t_{\text{tag}} \neq$ the $n_m - n_c$ most significant bits of x, go to 6
5. return t_{block} (**hit**)
6. read x from main memory (**miss**)

Example (**hit** for $x = 1001$)

Cache ($n_c = 2, n_b = 0$)

```
00 1 01 00010000
01 1 10 10000010
10 0 11 10000010
11 0 11 10010101
```

Main memory ($n_m = 4$)

```
0000 00000001
0001 00000010

1110 11000000
1111 11000001
```

1. $y = x_{[4-2,4[} = 01$
2. $t = [y] = [01] =$ `1 10 10000010`
   - $t_{\text{val}} = t_{[0,1[} = 1$
   - $t_{\text{tag}} = t_{[1,1+4-2[} = 10$
   - $t_{\text{block}} = t_{[1+4-2,1+4-2+8[} = 10000010$
3. $t_{\text{val}} = 1 = 1$
4. $t_{\text{tag}} = 10 = x_{[0,4-2[} = 10$

Example (**miss** for $x = 0000$)

Cache ($n_c = 2, n_b = 0$)

```
00 1 01 00010000
01 1 10 10000010
10 0 11 10000010
11 0 11 10010101
```

Main memory ($n_m = 4$)

```
0000 00000001
0001 00000010

1110 11000000
1111 11000001
```

1. $y = x_{[4-2,4[} = 00$
2. $t = [y] = [00] =$ `1 01 00010000`
3. $t_{\text{val}} = 1 = 1$
4. $t_{\text{tag}} = 01 \neq x_{[0,4-2[} = 00$
5. read 0000 from main memory
6. put `1 00 00000001` at $y = 00$
7. return $[x] =$ `00000001`

Miss rate and miss penalty

Given a sequence of n addresses $x_1, x_2, \ldots, x_n$

- the **miss rate** is the ratio between accesses resulting in a miss and n
  - the **hit rate** is $1-$ miss rate
- the **miss penalty** is the time taken to read a block from main memory and put it in the cache (steps 6 and 7)
  - during this time, the processor waits

Example of misses and hits

$x =$ 101, 000, 111, 001, 010 (left), 000, 101, 110, 111, 110 (right)

```
00 0 0 00000000
01 1 1 00100000
10 0 0 00000000
11 0 0 00000000

00 1 0 00000001
01 1 1 00100000
10 0 0 00000000
11 0 0 00000000

00 1 0 00000001
01 1 1 00100000
10 0 0 00000000
11 1 1 10000000

00 1 0 00000001
01 1 0 00000010
10 0 0 00000000
11 1 1 10000000

00 1 0 00000001
01 1 0 00000010
10 1 0 00000100
11 1 1 10000000
```

```
00 1 0 00000001
01 1 0 00000010
10 1 0 00000100
11 1 1 10000000

00 1 0 00000001
01 1 1 00100000
10 1 0 00000100
11 1 1 10000000

00 1 0 00000001
01 1 1 00100000
10 1 1 01000000
11 1 1 10000000

00 1 0 00000001
01 1 1 00100000
10 1 1 01000000
11 1 1 10000000

00 1 0 00000001
01 1 1 00100000
10 1 1 01000000
11 1 1 10000000
```

Initial cache

```
00 0 0 00000000
01 0 0 00000000
10 0 0 00000000
11 0 0 00000000
```

Main memory

```
000 00000001
001 00000010
010 00000100
011 00001000
100 00010000
101 00100000
110 01000000
111 10000000
```

hit rate $= 30\%$

Exploting spatial locality

Spatial locality: if a data location x is accessed at t, data locations *close* to x (some $x + \delta x$) will be likely accessed again soon

We need to have not only x but also $x + \delta x$ in cache
$\Rightarrow$ use larger blocks! (i.e., $s_b > 1$)

- miss penalty **increases too**: upon miss, larger data transfer from main memory

Direct mapping with $s_b > 1$

Assume $n_m = 6, n_c = 2, n_b = 2$ (block size is $s_b = 4$ bytes)

- block at $y = 00$ hosts
  - bytes from $x = 00\underline{00}00$ to $00\underline{00}11$
  - bytes from $x = 01\underline{00}00$ to $01\underline{00}11$
  - bytes from $x = 10\underline{00}00$ to $10\underline{00}11$
  - bytes from $x = 11\underline{00}00$ to $11\underline{00}11$
- block at $y = 01$ hosts
  - bytes from $x = 00\underline{01}00$ to $00\underline{01}11$
  - bytes from $x = 01\underline{01}00$ to $01\underline{01}11$
  - ...
- ...

Algorithm for reading x ($n_b \geq 0$)

1. $y = x_{[n_m - n_c - n_b, n_m - n_b[}$
2. $t = [y]$
   - $t_{\text{val}} = t_{[0,1[}$
   - $t_{\text{tag}} = t_{[1,1+(n_m-n_c-n_b)[}$
   - $t_{\text{block}} = t_{[1+(n_m-n_c-n_b),1+(n_m-n_c-n_b)+8\cdot 2^{n_b}[}$
3. if $t_{\text{val}} \neq 1$, go to 6
4. if $t_{\text{tag}} \neq x_{[0,n_m-n_c-n_b[}$, go to 6
5. return $t_{\text{block}[8z,8z+8[}$ with $z = x_{[n_m-n_b,n_m[}$ (**hit**) ($z = 0$ if $n_b = 0$)
6. read $x_0, \ldots, x_{s_b-1}$ from main memory (**miss**)
   - $x_0 = x_{[0,n_m-n_b[}$ 0…0 (n_b 0s)
   - $x_k = x_{[0,n_m-n_b[}$ k_2 (k as binary with n_b bits)
   - $x_{s_b-1} = x_{[0,n_m-n_b[}$ 1…1 (n_b 1s)

Example (**hit** for $x = 00001$)

Cache ($n_c = 2, n_b = 1$)

```
00 1 00 00000001 00000010
01 1 10 00100000 00110000
10 0 00 00000000 00000000
11 0 00 00000000 00000000
```

Main memory ($n_m = 6$)

```
00000 00000001
00001 00000010

11110 11100000
11111 11100001
```

1. $y = x_{[5-2-1,5-1[} = 00$
2. $t = [y] = [00] =$ `1 00 00000001 00000010`
   - $t_{\text{val}} = t_{[0,1[} = 1$
   - $t_{\text{tag}} = t_{[1,1+4-2[} = 00$
   - $t_{\text{block}} = t_{[1+4-2,1+4-2+8^2[} =$ `00000001 00000010`
3. $t_{\text{val}} = 1 = 1$
4. $t_{\text{tag}} = 00 = x_{[0,4-2[} = 00$

Misses and hits with $s_b = 4$

For a main memory of 256 byte, where $x = [x]$, an initially empty cache with 4 blocks of 1 word each, and the reads 10, 11, 13, 20, 21, 22, 10, 20, 21 (decimal addresses x)

- show the cache content after the 3rd request
- compute the hit rate

Hint: use decimal notation for block contents

("Initially empty" means all bits set to `0`)

Miss rate/penalty and block size

The larger the block size

- the lower the miss rate (for **better exploitment of spatial locality**)
- the longer the miss penalty
- the lower the number of blocks (for the same cache size)
  - thus, the greater the miss rate

Depends on the specific sequence of memory accesses

In practice

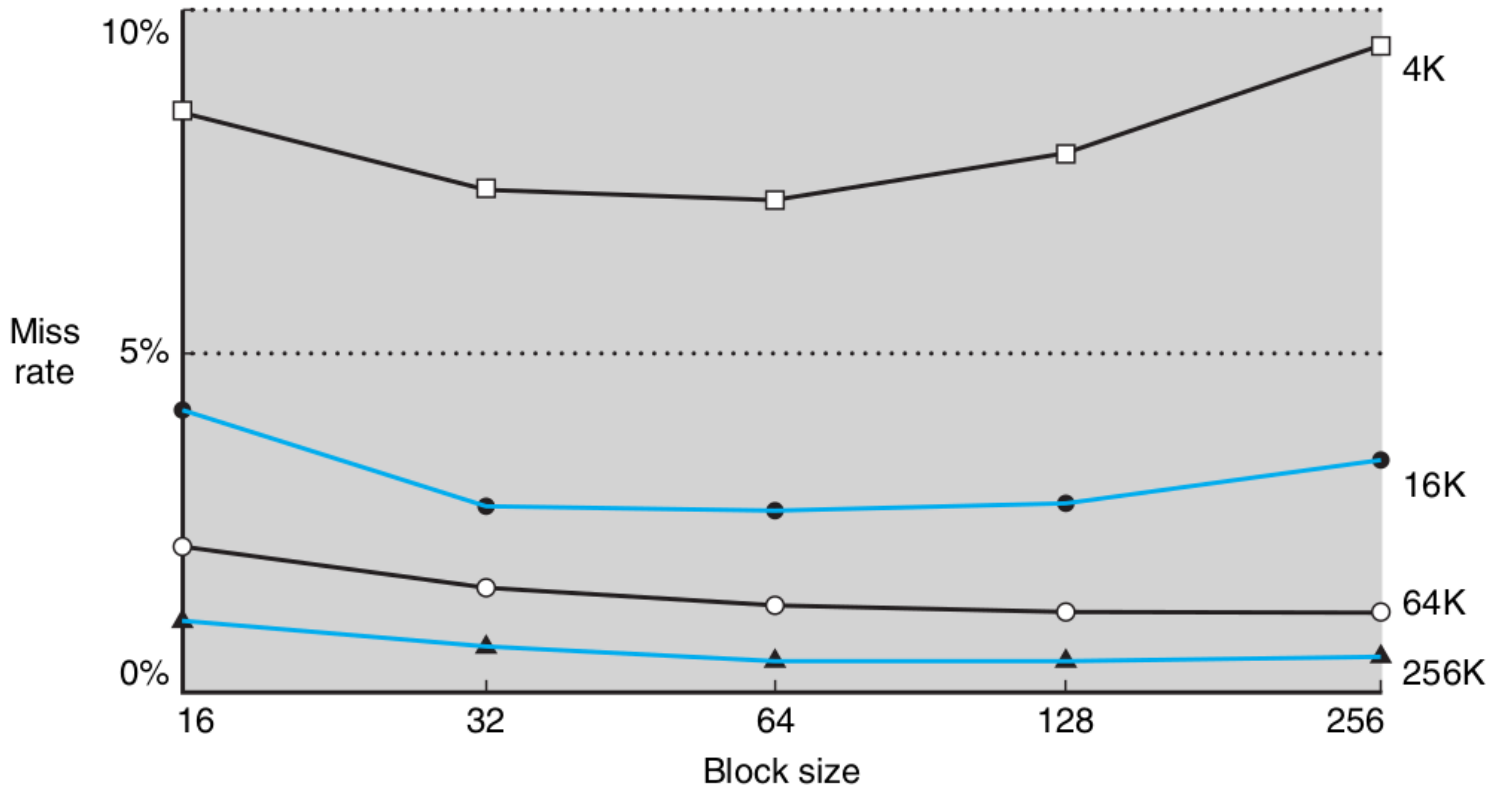


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so these data are based on SPEC92.

Miss penalty

The larger the block size

- the longer the miss penalty

Possible optimizations for reducing the penalty:

- **early restart**: restart when $[x]$ is read, not when the entire block is read from main memory
- **requested word first**: first read $[x]$, then *early restart*, then read the entire block

Miss penalty: numbers

Assume:

- $s_b = 4$
- 1 cycle per instruction (CPI)
- 1 cycle for reading one byte from cache
- 100 cycles for reading 4 bytes from main memory

Cycles for read:

- Hit: 1 cycle
- Miss: $4 \cdot 100 + 1 = 401$ cycles
- Miss with early restart: from $1 \cdot 100 + 1 = 101$ to $4 \cdot 100 + 1 = 401$ cycles, average 251 cycles
- Miss with requested word first: $1 \cdot 100 + 1 = 101$ cycles

Impact on CPI

- Hit: 1 cycle
- Miss: $4 \cdot 100 + 1 = 401$ cycles
- Miss with early restart: from $1 \cdot 100 + 1 = 101$ to $4 \cdot 100 + 1 = 401$ cycles, average 251 cycles
- Miss with requested word first: $1 \cdot 100 + 1 = 101$ cycles

Actual CPI based on miss rate:

	1%	**5%**	**10%**	**50%**
Miss	5	21	41	201
Miss with early restart	4	14	26	126
Miss with requested word first	2	6	11	51

Writes

If a write acts only on the cache, the content of the main memory at a given x and of the cache at the corresponding y may differ: they are **inconsistent**.

How to avoid inconsistencies?

1. write-through
2. write-back

Write-through

At *each* write:

- write at y in the cache **and**
- write at x in the main memory

Pros:

- "simple" strategy, both conceptually and for the implementation

Cons:

- basically cache does not give any advantage for writes

Impact on CPI

Assume:

- $s_b = 1$
- 1 cycle per instruction (CPI)
- 1 cycle for reading one byte from cache
- 100 cycles for reading/writing 1 byte from main memory

Actual CPI based on miss rate (y-axis) and read-to-write ratio (x-axis):

	20	**10**	**5**	**1**
1%	7	11	18	51
5%	10	15	22	53
10%	15	19	26	55
50%	53	55	59	75

Write-back

At each write:

- write at y in the cache

At each miss at y (read *and*[1] write): (1: can be optimized)

- first, write y block back on proper x
- then, load from x to y

Pros:

- more complex, harder to implement
  - more logic components, larger footprint, greater cost!

Cons:

Actual CPI

Consider a processor with a CPI of 2 and a miss penalty of 100 cycles for both read and write; consider a program with a 3-to-1 read-to-write ratio resulting in an overall 4% miss rate:

1. what's the change in actual CPI by halving the miss rate?
2. what's the change in actual CPI by halving the miss penalty?

(Three missing info: cycle to access cache; write policy; if not "actual r/w CPI", percentage of r/w instructions in the program)

A real cache (FastMATH processor)

Decreasing miss rate

One option: increasing block size

- has some cons

Another option: each x can be hosted in (*associated* with) many y, not in just one.
→ increase the degree of **associativity**

[What's the counterpart of associativity in the library case?]

Associativity

- each x in exactly one y: 1-way associativity
  - i.e., no associativity
- each x in n y: n-way associativity
  - usually, $n = 2^k$ (in practice, 2 or 4)
- each x in all y: full associativity
  - $k = n_c$

With $k = 1$: many-to-one from x to y

With $k > 1$: many-to-many from x to y

Finding/putting an x with > 1-associativity

Where should I look in the cache for a given x?

- with 1-way: in exactly one y
- with n-way: in n y

Where should I put an x in the cache?

- with 1-way: in exactly one y
- with n-way: in exactly one y **chosen among** n; how?
  - least recently used (LRU)
  - randomly

Set of blocks

With associativity, s_c blocks are organized in **sets of blocks**, each consisting of $s_s = 2^{n_s}$ contiguous blocks (former n becomes s_s)

- in practice, $s_s \in 1, 2, 4, s_c$

A given x will go in a block of the k-th set, with $x \ \% \ \frac{s_c}{s_s} = k$
(instead of the k-th block)

- $\frac{s_c}{s_s}$ is the number of sets of blocks, it takes $n_c - n_s$ bits

Tag indicates which x is in a given y

x to y

Assume $n_s = 2, n_c = 4$:

Block index k to y:

- set $k = \texttt{00}$ contains blocks from $y = \underline{\texttt{00}}\texttt{00}$ to $\underline{\texttt{00}}\texttt{11}$
- set $k = \texttt{01}$ contains blocks from $y = \underline{\texttt{01}}\texttt{00}$ to $\underline{\texttt{01}}\texttt{11}$
- set $k = \texttt{10}$ contains blocks from $y = \underline{\texttt{10}}\texttt{00}$ to $\underline{\texttt{10}}\texttt{11}$
- set $k = \texttt{11}$ contains blocks from $y = \underline{\texttt{11}}\texttt{00}$ to $\underline{\texttt{11}}\texttt{11}$

x to k: $x \,\%\, \frac{s_c}{s_s} = k$, that is, $k = x_{[n_m-(n_c-n_s)-n_b, n_m-n_b[}$

Hence, $x \to Y = \{y : y_{[0,n_c-n_s[} = x_{[n_m-(n_c-n_s)-n_b, n_m-n_b[}\}$

(n_c "becomes" $n_c - n_s$)

Algorithm for reading x

1. $Y = \{y : y_{[0,n_c-n_s[} = x_{[n_m-(n_c-n_s)-n_b,n_m-n_b[}\}$
2. **for each** $y \in Y$
    1. $t = [y] = (t_{\text{val}}, t_{\text{tag}}, t_{\text{block}})$
    2. if $t_{\text{val}} = 1$ and $t_{\text{tag}} = x_{[0,n_m-(n_c-n_s)-n_b[}$
    3. return $t_{\text{block}[8z,8z+8[}$ with $z = x_{[n_m-n_b,n_m[}$ (**hit**) (0 if $n_b = 0$)
3. read $x_0, \ldots, x_{s_b-1}$ from main memory (**miss**)
4. **choose** k (with n_s bits)
5. put 1 $x_{[0,n_m-(n_c-n_s)-n_b[}$ $[x_0] \ldots [x_{s_b-1}]$ at $y = x_{[n_m,n_m-(n_c-n_s)-n_b[}$ k_2
6. return $[x]$

Steps 2.1 to 2.3 are actually done in parallel!

Example with misses and hits

$n_c = 3, n_s = 1, n_m = 8, n_b = 1$

```
000 1 00000 00000001 00000010
001 1 01010 00100000 00110000
010 0 11010 00000000 00000000
011 1 01111 00000000 11100000
100 1 00101 00000001 00000010
101 1 10110 00100000 00110000
110 1 01100 00000000 00000000
111 0 01011 00000000 00000000
```

$x = 00100110 \Rightarrow Y = \{110, 111\} \Rightarrow$ miss (tag), miss (validity)

$x = 01111011 \Rightarrow Y = \{010, 011\} \Rightarrow$ miss (validity), hit

$\Rightarrow$ returns $t_{\text{block}[8z,8z+8[} = 11100000$ with $z = x_{[8-1,8[} = 1$

$x = 01011000 \Rightarrow Y = \{000, 001\} \Rightarrow$ miss (tag), miss (tag)

[Can we have > 1 hits in a block?]

Choosing a block in set (LRU)

For each block in the set, we need to know *when* it was used last time: it can be done with n_a recency bits (a *recency tag*):

- `00` for the most recently used
- `01` for the 2nd most recently used
- `10` for the 3rd third most recently used
- `11` for the 4th most recently used

Whenever an y is accessed, if its recency tag is $\neq$ `00`, the recency tags of **all blocks in the set** have to be updated:

- the one at this y becomes `00`
- each other is increased by 1 (`11` stays `11`)

Choosing a block in set (random)

Just random!

- no recency tags needed
- no update on hits

Much less complex than LRU:

- in practice, LRU is used only with $n_s = 1$ or 2
- impact on miss rate is low: $\approx 10\%$
  - negligible with large caches

Misses and hits with associativity

For a main memory of 256 byte, where $x = [x]$, an initially empty **associative** cache with 4 blocks of 1 word each, and the reads 10, 11, 13, 20, 21, 22, 10, 20, 21 (**hexadecimal** addresses x):

- compute the percentage improvement in hit rate for $s_s = 2$ with respect to the case of no associativity
- compute the overall size of the cache for $s_s = 2$ and LRU

("Initially empty" means all bits set to `0`)

Virtual memory

Goals

Goal of cache: memory with fast access and large size

Unsolved issues:

1. memory used by many programs at the same time
2. physical memory being smaller than addressable memory

Memory sharing

Suppose program A and program B are being executed **concurrently**: (we'll see, briefly, how)

Question:

- how to prevent A from accessing an x address that "belongs" to B?

Trivial, but *wrong*, solution: make A access addresses from $x_{A,i}$ to $x_{A,f}$ and B access from $x_{B,i}$ to $x_{B,f}$, with $[x_{A,i}, x_{A,f}]$ and $[x_{B,i}, x_{B,f}]$ being disjoint...

- works only for A and B, what about C, D, ...
- works only if $[x_{A,i}, x_{A,f}]$ is free (same for B)
- require knowing $[x_{A,i}, x_{A,f}]$ at compile time

Addressable memory

In ancient times, RAMs were small, much smaller than the addressable space (i.e., 2^{n_m})

Today, RAMs are much bigger, but still often smaller than addressable space:

- $n_m = 32$ means 4 GB: not all computers have 4 GB RAM
- $n_m = 64$ means 18 exabytes... (actually, depends on the architecture: usually "much" lower)

Questions:

- how to allow programs to use a memory of 2^{n_m} bytes having a physical memory that is smaller?
- (further) how to make all running programs "see" a memory of size 2^{n_m}?

Idea (sketch of)

At any time, the memory content is only partially in the physical memory (RAM); the remaining part is on the disk

When a program wants to access a given address, it may be on the physical memory:

- if yes, nice
- if not, find it on the disk, bring it to the physical memory, and use it

Solves the issue of size, not that of concurrent access to memory

Idea (finer sketch of)

Memory is organized in **pages**:

- each page has a **page number**
- within page, bytes has an address (**page offset**) starting at 0
- a physical memory location is composed of a page number *and* a page offset

Each program is given a page and accesses memory specifying the page offset

Solves the issues of concurrent access, not that of size

- A cannot access the page of B
- A is compiled specifying just the page offset, the page number at runtime is assigned by *someone*

Idea (together)

- memory is organized in pages
- programs are given **many** pages, not just one
  - from program POV, the first has always number 0
- at any time a page may be in the physical memory or on the disk
  - programs do not need to know if a page is on memory or on disk

Solves all problems:

- there can be *as many pages as we want*, actually a number of pages that is *like* 2^{n_m} bytes for each program
- no undesired interference among programs (there can be regulated sharing of page, though)

Virtual memory and addresses

From program point of view, a memory address is given by $\langle$page number, page offset$\rangle$

- each program has pages starting from number 0

But physically, bytes are either in RAM or on disk

- in general, not starting "at 0"

$\Rightarrow$ addresses has to be **translated**

Virtual memory and caches

There are similarities:

- both motivated by "have your cake and eat it too"
- both exploit a memory hierarchy (i.e., two memories)
- both are organized in chunks
- both require an address translation

	Cache	**Virtual memory**
Goal	fast as SRAM, large as DRAM	private as split for programs, large as entire for each program
Two memories	SRAM and DRAM	DRAM and disk
Chunks	block	page
Address translation	x to y (or Y)	we'll see

Address translation

In cache:

- x is the physical address in the (main) memory (in general, the lower level memory)
- y is the physical block address in the cache

In virtual memory:

- x_v is the **virtual address**
  - generated by the program
- x_p is the **physical address**

Address parts

Assume:

- a virtual memory addressable with $n_v = 32$ bits
- a page size of $s_p = 16$ KB, i.e., $n_p = \log_2(16 \cdot 1024) = 4 + 10 = 14$

A virtual address is composed of:

- the page number, ranging from 0 to $s_n = \frac{2^{32}}{2^{14}}$, i.e., $n_n = n_v - n_p = 18$

In general, the virtual memory can be larger than the physical memory (*illusion* of large memory):

- $n_v > n_m$, e.g., $n_v = 48$ and $n_m = 40$

Finding a page

Assume a program wants to access x_v: what happens?

In brief and conceptually:

1. get page number $p_v = x_{v[0,n_n[}$ from x_v
2. if page p is in physical memory
   - translate x_v to x_p
   - access it
3. otherwise (**page fault**)
   - find it on disk (where?)
   - move it in memory (where? remove what?)
   - translate x_v to x_p
   - access it

Looks very like the case of cache, but...

Cost of miss vs. fault

Type	Access time [ns]	Cost [$/GB]
SRAM	0.5–2.5	500–1000
DRAM	50–70	10–20
Flash	5000–50000	0.75–1.00
Magnetic disk	5000000–20000000	0.05–0.10

Cache:

- hit (SRAM): 1 cycle
- miss (DRAM): $\approx \frac{50}{0.5} = 100$ cycles, $100\times$ longer

Virtual memory:

- hit (DRAM): $t \approx 50$
- fault (disk): $t \approx 5 \cdot 10^3$, $100000\times$ longer

Cost of page fault

Since the cost of page faults is very high, greater care is put in reducing page fault probability:

- full associativity
  - with smarter replacement strategies employed by OS (operating system)
- much larger chunk size
  - exploits high bandwidth, despite big latency, of disks
- write back
  - write through would be very unpractical ($100000\times$ longer)

Page table

Every program has a **page table** consisting of 2^{n_n} entries

Each entry contains:

- one validity bit
- a physical page number p_p of $n_m - n_p$ bits (the n_p bits are the page offset)

The translation from x_v to x_p is a look-up in the page table:

1. take $p_v = x_{v[0,n_n[}$
2. look at p_v-th entry in the page table
3. if the validity bit is set (**hit**), $x_p = p_p x_{v[n_n,n_v[}$
4. otherwise (**miss**) ...

Program state

For a program, the page table is in memory

- hardware is optimized for keeping the address of the page table (that is accessed frequently) in a specific registry

State of a program (also called **process**):

- program counter
- registers
- page table (address)

State can be stored (in memory) and restored back:

- **active** process if contents above are in place
- **inactive** process otherwise

Page location on fault

The page table does not contain the physical address of the faulted page on the disk

The OS takes care of reserving a disk portion for *all* the pages of each project: **swap space** (actually, things can be more complex)

- usually pages for a program are contiguous on swap space
- p_v can be easily translated to a position on the swap disk for a given program

OS takes care of page faults!

- they are long enough to make convinient the management by sw instead of hw

(first request for a page does not need to go on disk for reading)

Replacement policy

Because of full associativity, any page table entry can be replaced

Since replacement is managed by OS, replacement policy may be complex (e.g., LRU may be preferable over random)

However, real LRU is still too costly, since require updating usage info at each access, not only upon faults

$\Rightarrow$ *approximated* LRU with usage bit

Usage/reference bit

Each entry in the page table contains:

- one validity bit
- one **usage** or reference bit
- a physical page number p_p

At every hit, the usage bit is set

Every while, **all** usage bits are unset

(for bits, "set" is "set to 1", "unset" is "set to 0")

[How to estimate the effectiveness of dirty bit with respect to actual LRU?]

Size of page table

The page table may be **very large**!

E.g., for $n_v = 48$, $n_p = 14$, $n_m = 40$, there are 2^{n_n} entries, each consisting of $1 + 1 + n_m - n_p = 28$ bits, with $n_n = n_v - n_p = 48 - 14 = 34$
$\Rightarrow 28 \cdot 2^{34} \approx 60$ GB!!! one per each process!!!

Unfeasible to take them in memory!

Optimizations:

- use smaller tables (process address a smaller virtual memory)
- use dynamic tables (initially small, grown on need)
  - possibly in two directions for heap/stack growing requests
- hash the the virtual page numbers (*inverted page table*)

Writes

Write back policy with optimization: for avoding writing back when unneeded (because the page has not be written)

Each entry in the page table contains:

- one validity bit
- one usage bit
- one **dirty** bit
- a physical page number p_p

Upon read/creation, the dirty bit is unset

Upon writes on the page, the dirty bit is set