Large and Fast: Memory Hierarchy [7]

- Principle of Locality
- Memory Technology
- Memory Hierarchy Structure
- Caches
- Virtual Memory

chapter7.ps.zip

Principle of Locality [7.1]

More

Memory systems are organized as a memory **hierarchy** because of the **principle of locality**.

Principle of Locality [7.1.1]

The principle of locality has two components:

- **temporal locality**—If location X is accessed, it is likely to be accessed **again** in the near future.
- **spatial locality**—If location X is accessed, other locations **close to X** are likely to be accessed in the near future.

Principle of Locality [7.1.2]

Most software exhibits these characteristics:

- **Instruction accesses**
  - spatial locality due to sequential instruction access
  - temporal locality due to loops and recursion
- **Data accesses**
  - spatial locality due to arrays and structures
  - temporal locality due to temporaries

Memory Technology [7.2]

More

There are a variety of memory technologies. Each provides a different blend of **speed** (access time) and **cost per bit**

Memory Technology [7.2.1]

The three most common technologies are

- **SRAM**—Static RAM
  Transistor feedback cells (no refreshing)
  Access time: 1 to 10 ns
- **DRAM**—Dynamic RAM
  Capacitor cells (requires refreshing)
  Access time: 10 to 100 ns
- **Magnetic Disk**
  Mechanical arm and platters
  Access time: 10 to 50 ms

Memory Technology [7.2.2]

Note that magnetic disks are a **million** times slower than DRAM: ms = 10^6 X ns.

Also, memory speed is directly proportional to cost:

SRAM

DRAM

Disk

↑ **increasing speed** (decreasing access time)

↑ **increasing cost** (in terms of $ per MByte)

Memory Technology [7.2.3]

If you can spend a fixed amount of money on only **one memory technology**, then you can buy

large and **slow** **OR** **small** and **fast**

However, you want **large** and **fast**.

Memory Hierarchy Structure
[7.3]

More

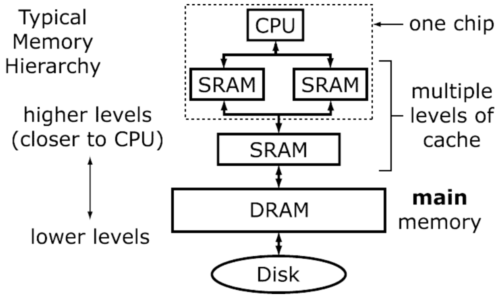
If the principle of locality holds, we can combine memory technologies to give the illusion of a **large** and **fast** memory.

Memory Hierarchy Structure
[7.3.1]

How can combining memory technologies help?

1. When a location is accessed the **first** time, retrieve the value from long-term storage in a large and slow memory.
2. However, **save** the value along with the values of nearby locations in a small and fast memory.
3. Accesses to these values **are very likely in the near future** and they can be satisfied by the small and fast memory.

Memory Hierarchy Structure
[7.3.2]



Memory Hierarchy Structure
[7.3.3]

Common terminology:

- **hit**—a memory access that finds the value in the current memory level
- **miss**—a memory access that does not find the value in current memory level; the value must be sought in lower levels
- **hit time**—access time if a hit occurs
- **miss penalty**—additional access time if a miss occurs (can vary)

Caches
[7.4]

Cache Overview

Cache Organization

Example Cache Operation

Caches and Writing

Miss Analysis

Cache Overview
[7.4.1]

More

Caches are **high-speed, temporary storage** for memory values likely to be accessed in the near future.

Cache Overview
[7.4.1.1]

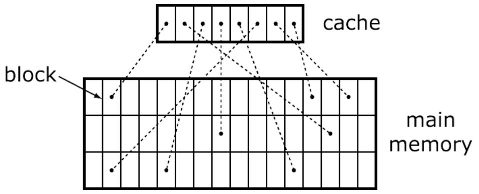
Caches are **small** (compared to main memory), high-speed **SRAM** memories that are close to the CPU.

By satisfying most memory requests, caches decrease the average memory access time.

A **block** or **cache line** is the unit of data transferred between a cache and the next lower memory level. Block sizes are typically between 4 and 256 bytes.

Cache Overview
[7.4.1.2]

A cache stores blocks from widely scattered locations—how are the blocks organized?



First, we need to know about block addressing.

Cache Overview [7.4.1.3]

Every byte in main memory has a **byte address**. Once a block size is chosen, every main memory byte also has a **block address**:

$$\text{block address} = \left\lfloor \frac{\text{byte address}}{\text{block size in bytes}} \right\rfloor$$

floor function (i.e., drop the fraction)

Example:

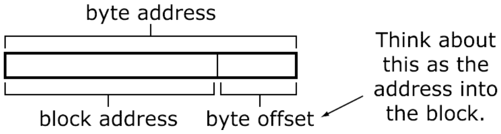
byte address = 963

block size = 16 bytes

$$\text{block address} = \left\lfloor \frac{963}{16} \right\rfloor = \lfloor 60.188 \rfloor = 60$$

Cache Overview [7.4.1.4]

When the block size is a **power of two**, the byte address divides cleanly into two parts:

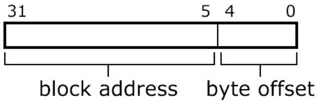


Think about this as the address into the block.

(Note that block addresses are a generalization of what we have called "word addresses", "halfword addresses"etc.)

Cache Overview [7.4.1.5]

Example:
byte addr = 32 bits
block size = 32 bytes (or 8 words)
Thus, the byte offset will be 5 bits since 32 = 2^5.
block address = 32 - 5 = 27 bits



Cache Organization [7.4.2]

Fully Associative Caches

Direct Mapped Caches

Set Associative Caches

Block Replacement Policy

Fully Associative Caches [7.4.2.1]

More

With a **fully associative** organization, blocks from main memory can be placed **anywhere** in the cache.

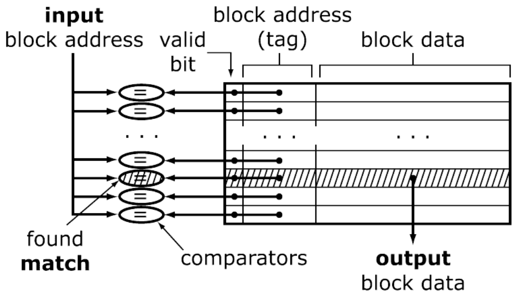
Fully Associative Caches [7.4.2.1.1]

Fully associative caches get their name from **associative memories**.

Associative memories use **comparators** to retrieve a value based on the value's **content** instead of an index into the memory.

In a fully associative cache, blocks are stored along with their block address. **Blocks are located by searching for the block with a given block address**.

Fully Associative Caches [7.4.2.1.2]



Fully Associative Caches [7.4.2.1.3]

However, memory density is reduced because the comparators require **a lot** of hardware.

Also, building the comparators into hardware tends to reduce flexibility in address and block size.

We will find that other cache approaches work almost as well, so associative memories are **low volume, expensive** parts and **large, fully associative caches are rare**.

Direct Mapped Caches
[7.4.2.2]

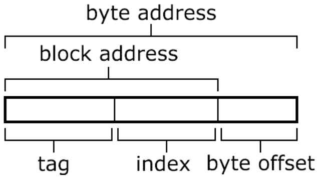
More

By restricting the placement of blocks, the **direct mapped** approach needs only **one comparator**.

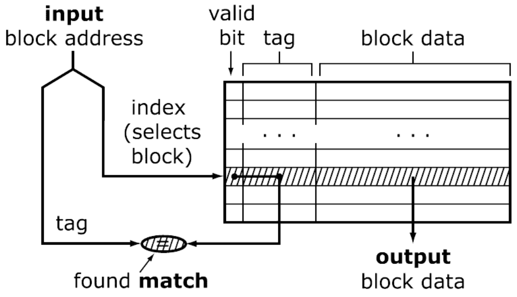
Direct Mapped Caches
[7.4.2.2.1]

To reduce the number of comparators, blocks can be placed into the cache according to an **index** value derived from the block address.

The simplest **index** is just some lower bits of the block address. The remaining bits are called the **tag**.



Direct Mapped Caches
[7.4.2.2.2]



Direct Mapped Caches
[7.4.2.2.3]

A block can go into **only one location**—the one identified by its index.

That is, the index is the **address** of the block in the cache. If we let

indexCount = number of index bits

blockCount = number of cache blocks

then

$2^{\text{indexCount}}$ = blockCount

index = (block address) mod blockCount

Direct Mapped Caches
[7.4.2.2.4]

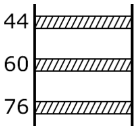
Example: block address = 60, blockCount = 16

index = 60 mod 16 **= 12**

Problem: Different blocks may have the same index (just like hashing)—called **interference** or **conflict**.

Example:

If blockCount = 16, then all of these blocks have the same index (12):



Set Associative Caches
[7.4.2.3]

More

Set associative caches are a **compromise** between fully associative and direct mapped caches.

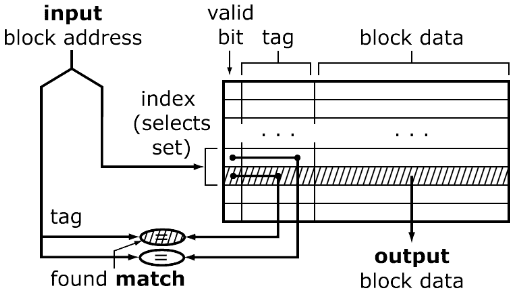
Set Associative Caches
[7.4.2.3.1]

Set associative caches use a **small number of comparators** (like direct mapped), but they retain **flexibility in block placement** (like fully associative).

A set associative cache is **divided into sets**. "**n-way set associative**" means that there are n blocks in each set.

The **index is used to select the set** instead of the block.

Set Associative Caches
[7.4.2.3.2]



Set Associative Caches [7.4.2.3.3]

A block can be placed anywhere **within the set** that its index selects.

Note these special cases:

- direct mapped is "1-way set associative"
- fully associative is "(blockCount)-way set associative"

Also, if setCount = number of sets in cache
index = (block address) mod setCount

Block Replacement Policy [7.4.2.4]

More

When the possible cache locations for placing a new block are already filled, some block must be chosen for replacement.

Block Replacement Policy [7.4.2.4.1]

When a new block is brought into the cache, where does it go?

- **direct mapped**—only **one location**
  If a block is already there, replace it.
- **fully or set associative**—**multiple locations** possible
  If there are already blocks in all of those locations, one must be chosen for replacement—which one?

Block Replacement Policy [7.4.2.4.2]

Choosing a block to replace:

- **random**—pick a block at random; actually works quite well in practice.
- **least recently used (LRU)**—pick the block that has not been accessed for the longest time

LRU is hard to do **exactly** in hardware, so there are various approximation schemes.

Example Cache Operation [7.4.3]

Direct Mapped Example

Set Associative Example

Fully Associative Example

Direct Mapped Example [7.4.3.1]

More

Consider the operation of a direct mapped cache through a series of 12 memory accesses.

Direct Mapped Example [7.4.3.1.1]

Assume the cache has a total of 32 words and a block size of 2 words.

32/2 = **16 blocks** in the cache
2^4 = 16, so the **index is 4 bits**
2^3 = 8 bytes, so the **byte offset is 3 bits**
32 - 4 - 3 = 25, so the **tag is 25 bits**

Total cache size is
blockCount X (valid bit + tag + data)
= 16(1 + 25 + 64) = 1,440 bits

Direct Mapped Example [7.4.3.1.2]

For cache index calculations:

$$\text{block address} = \left\lfloor \frac{\text{byte address}}{8} \right\rfloor$$

index = (block address) mod 16

Direct Mapped Example [7.4.3.1.3]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
580	72	8	M

Updated Cache Contents

Index	Block
0	
1	
2	
3	
4	
5	
6	
7	
8	72
9	
10	
11	
12	
13	
14	
15	

Direct Mapped Example [7.4.3.1.4]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
312	39	7	M

Updated Cache Contents

Index	Block
0	
1	
2	
3	
4	
5	
6	
7	39
8	72
9	
10	
11	
12	
13	
14	
15	

Direct Mapped Example [7.4.3.1.5]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
580	72	8	H

Updated Cache Contents

Index	Block
0	
1	
2	
3	
4	
5	
6	
7	39
8	72
9	
10	
11	
12	
13	
14	
15	

Direct Mapped Example [7.4.3.1.6]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
364	45	13	M

Updated Cache Contents

Index	Block
0	
1	
2	
3	
4	
5	
6	
7	39
8	72
9	
10	
11	
12	
13	45
14	
15	

Direct Mapped Example
[7.4.3.1.7]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
536	67	3	M

Updated Cache Contents

Index	Block
0	
1	
2	
3	67
4	
5	
6	
7	39
8	72
9	
10	
11	
12	
13	45
14	
15	

Direct Mapped Example
[7.4.3.1.8]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
324	40	8	M

Updated Cache Contents

Index	Block
0	
1	
2	
3	67
4	
5	
6	
7	39
8	~~72~~ 40
9	
10	
11	
12	
13	45
14	
15	

Direct Mapped Example
[7.4.3.1.9]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
412	51	3	M

Updated Cache Contents

Index	Block
0	
1	
2	
3	~~67~~ 51
4	
5	
6	
7	39
8	~~72~~ 40
9	
10	
11	
12	
13	45
14	
15	

Direct Mapped Example
[7.4.3.1.10]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
528	66	2	M

Updated Cache Contents

Index	Block
0	
1	
2	66
3	~~67~~ 51
4	
5	
6	
7	39
8	~~72~~ 40
9	
10	
11	
12	
13	45
14	
15	

Direct Mapped Example [7.4.3.1.11]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
664	83	3	M

Updated Cache Contents

Index	Block
0	
1	
2	66
3	~~67~~ ~~51~~ 83
4	
5	
6	
7	39
8	~~72~~ 40
9	
10	
11	
12	
13	45
14	
15	

Direct Mapped Example [7.4.3.1.12]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
576	72	8	M

Updated Cache Contents

Index	Block
0	
1	
2	66
3	~~67~~ ~~51~~ 83
4	
5	
6	
7	39
8	~~72~~ ~~40~~ 72
9	
10	
11	
12	
13	45
14	
15	

Direct Mapped Example [7.4.3.1.13]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
668	83	3	H

Updated Cache Contents

Index	Block
0	
1	
2	66
3	~~67~~ ~~51~~ 83
4	
5	
6	
7	39
8	~~72~~ ~~40~~ 72
9	
10	
11	
12	
13	45
14	
15	

Direct Mapped Example [7.4.3.1.14]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
536	67	3	M

Updated Cache Contents

Index	Block
0	
1	
2	66
3	~~67~~ ~~51~~ ~~83~~ 67
4	
5	
6	
7	39
8	~~72~~ ~~40~~ 72
9	
10	
11	
12	
13	45
14	
15	

Note
misses
d

due to
conflict

Direct Mapped Example [7.4.3.1.15]

Note that for this example

$$\text{miss rate} = \frac{\text{number of misses}}{\text{number of accesses}} = \frac{10}{12} = 83.3\%$$

Set Associative Example [7.4.3.2]

More

Consider the operation of a set associative cache of the same size as it processes the same series of 12 memory accesses.

Set Associative Example [7.4.3.2.1]

Assume the cache is **2-way** set associative, a total of 32 words, a block size of 2 words, and LRU replacement.

Still **16 blocks**, but now 16/2 = **8 sets**
2^3 = 8, so the **index is 3 bits**
Same block size means **byte offset is 3 bits**
32 - 3 - 3 = 26, so the **tag is 26 bits**

Total cache size is
= 16(1 + 26 + 64) = 1,456 bits

Set Associative Example [7.4.3.2.2]

For cache index calculations:

$$\text{block address} = \left\lfloor \frac{\text{byte address}}{8} \right\rfloor$$

$$\text{index} = (\text{block address}) \bmod 8$$

Set Associative Example [7.4.3.2.3]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
580	72	0	M

Updated Cache Contents

Index	Block A	Block B
0	72	
1		
2		
3		
4		
5		
6		
7		

Set Associative Example [7.4.3.2.4]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
312	39	7	M

Updated Cache Contents

Index	Block A	Block B
0	72	
1		
2		
3		
4		
5		
6		
7	39	

Set Associative Example [7.4.3.2.5]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
580	72	0	H

Updated Cache Contents

Index	Block A	Block B
0	72	
1		
2		
3		
4		
5		
6		
7	39	

Set Associative Example [7.4.3.2.6]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
364	45	5	M

Updated Cache Contents

Index	Block A	Block B
0	72	
1		
2		
3		
4		
5	45	
6		
7	39	

Set Associative Example [7.4.3.2.7]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
536	67	3	M

Updated Cache Contents

Index	Block A	Block B
0	72	
1		
2		
3	67	
4		
5	45	
6		
7	39	

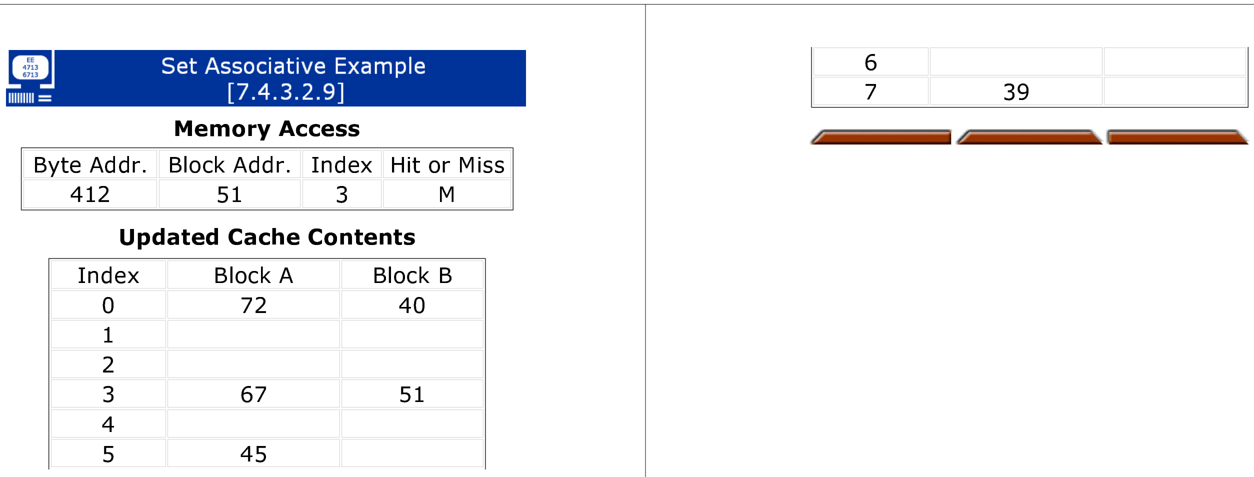
Set Associative Example [7.4.3.2.8]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
324	40	0	M

Updated Cache Contents

Index	Block A	Block B
0	72	40
1		
2		
3	67	
4		
5	45	
6		
7	39	



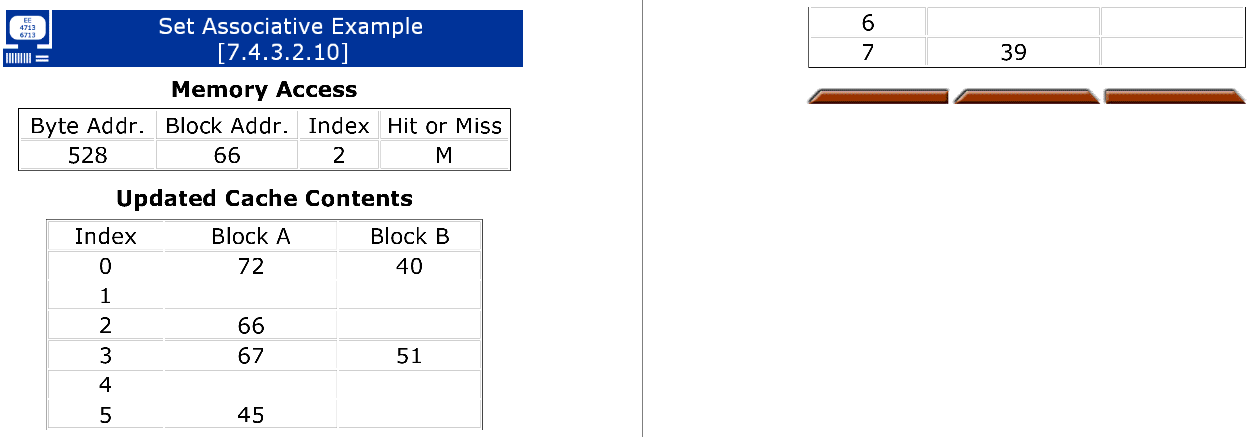
Set Associative Example [7.4.3.2.9]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
412	51	3	M

Updated Cache Contents

Index	Block A	Block B
0	72	40
1		
2		
3	67	51
4		
5	45	
6		
7	39	



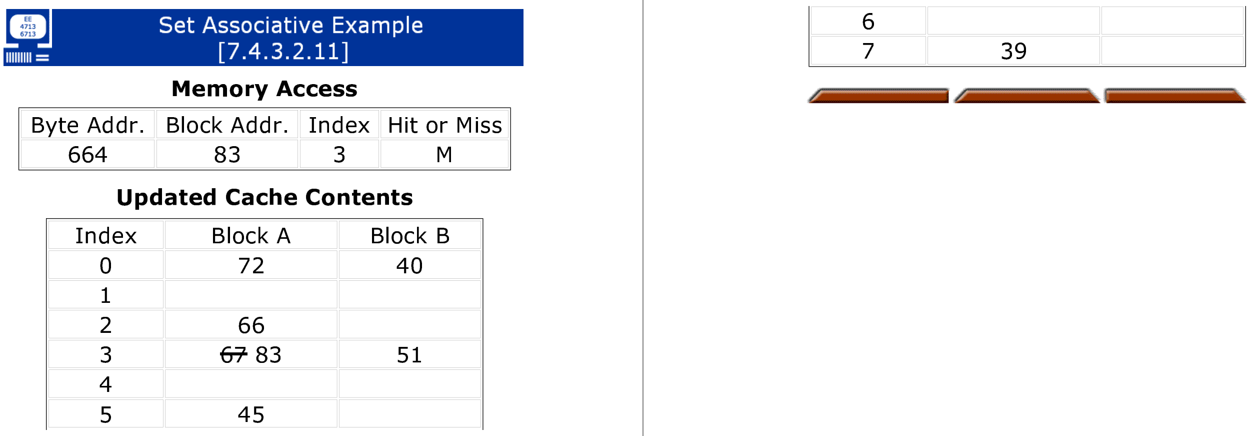
Set Associative Example [7.4.3.2.10]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
528	66	2	M

Updated Cache Contents

Index	Block A	Block B
0	72	40
1		
2	66	
3	67	51
4		
5	45	
6		
7	39	



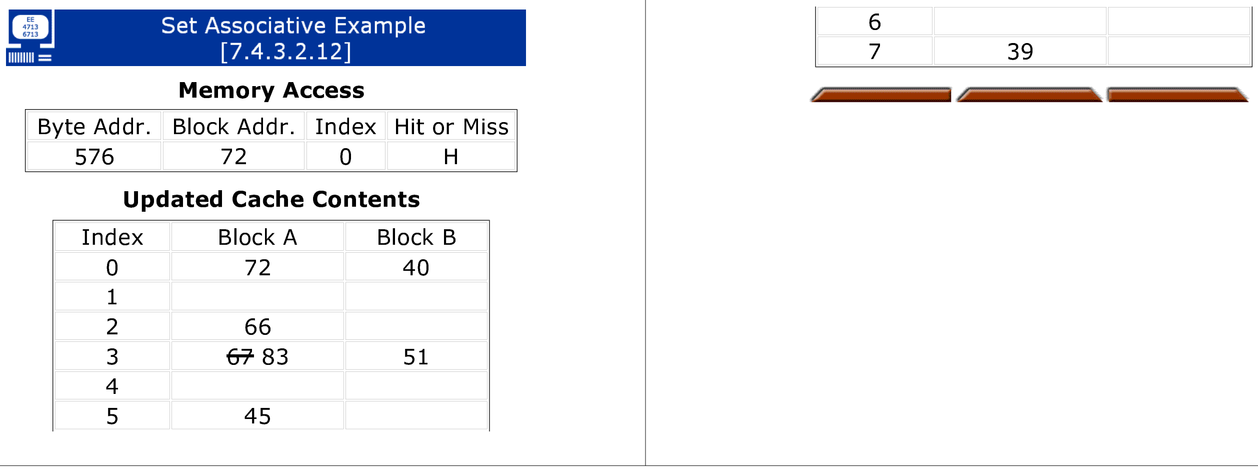
Set Associative Example [7.4.3.2.11]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
664	83	3	M

Updated Cache Contents

Index	Block A	Block B
0	72	40
1		
2	66	
3	~~67~~ 83	51
4		
5	45	
6		
7	39	



Set Associative Example [7.4.3.2.12]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
576	72	0	H

Updated Cache Contents

Index	Block A	Block B
0	72	40
1		
2	66	
3	~~67~~ 83	51
4		
5	45	
6		
7	39	

Set Associative Example [7.4.3.2.13]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
668	83	3	H

Updated Cache Contents

Index	Block A	Block B
0	72	40
1		
2	66	
3	~~67~~ 83	51
4		
5	45	
6		
7	39	

Set Associative Example [7.4.3.2.14]

Memory Access

Byte Addr.	Block Addr.	Index	Hit or Miss
536	67	3	M

Updated Cache Contents

Index	Block A	Block B
0	72	40
1		
2	66	
3	~~67~~ 83	~~51~~ 67
4		
5	45	
6		
7	39	

Set Associative Example [7.4.3.2.15]

Note that for this example

$$\text{miss rate} = \frac{\text{number of misses}}{\text{number of accesses}} = \frac{9}{12} = 75\%$$

The miss rate improved with the increase in associativity.

Fully Associative Example [7.4.3.3]

More

Consider the operation of a fully associative cache of the same size as it processes the same series of 12 memory accesses.

Fully Associative Example [7.4.3.3.1]

Assume the cache is fully associative, a total of 32 words, a block size of 2 words, and LRU replacement.

Still **16 blocks**
No index bits
Same block size means **byte offset is 3 bits**
32 - 3 = 29, so the **tag is 29 bits**

Total cache size is
= 16(1 + 29 + 64) = 1,504 bits

Fully Associative Example [7.4.3.3.2]

Memory Accesses

Byte Addr.	Block Addr.	Hit or Miss
580	72	M
312	39	M
580	72	H
364	45	M
536	67	M
324	40	M
412	51	M
528	66	M

Final Cache Contents
(Letters are block labels)

A	B	C	D
72	39	45	67
E	F	G	H
40	51	66	83
I	J	K	L
M	N	O	P

664	83	M
576	72	H
668	83	H
536	67	H

miss rate = 8/12 = 67%

Greater associativity improves the miss rate.

Caches and Writing [7.4.4]

More

When a store is executed, we have to make sure that the written value finds its way back to main memory.

Caches and Writing [7.4.4.1]

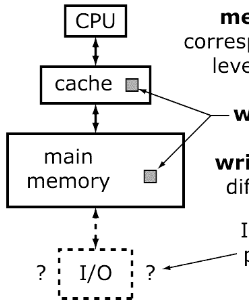
There are two aproaches to handling a store:

- **write through**—Writes are much less frequent than reads so just write into cache and main memory.
- **write back**—Write only to cache and use "dirty" bit to keep track of which cache blocks have been changed.
  Copy to main memory when a dirty block is replaced.

Caches and Writing [7.4.4.2]

	Adv.	Disadv.
Write Through	simple and memory stays consistent	slower (write buffers can help)
Write Back	faster and consolidates writes	more complex controller and memory may be inconsistent at times

Caches and Writing [7.4.3]



memory inconsistency: corresponding blocks at different levels have different values

write through: blocks stay the same.
write back: blocks can be different for a long time.

Inconsistency is not a problem until we add other "observers".

Miss Analysis [7.4.5]

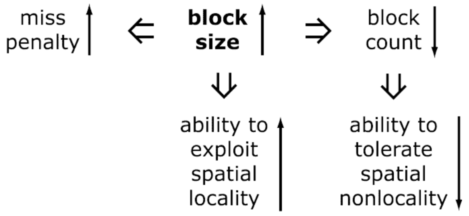
- Block Size and Misses
- Associativity and Misses
- Miss Classification
- Reducing the Miss Penalty

Block Size and Misses [7.4.5.1]

More

For a given cache size, there is an **optimal block size** for a program. Blocks that are too small do not exploit spatial locality, but blocks that are too large cannot tolerate spatial nonlocality.

Block Size and Misses [7.4.5.1.1]



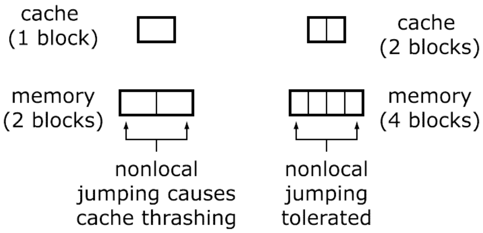
miss penalty ↑ ⇐ **block size** ↑ ⇒ block count ↓

⇓ ability to exploit spatial locality ↑

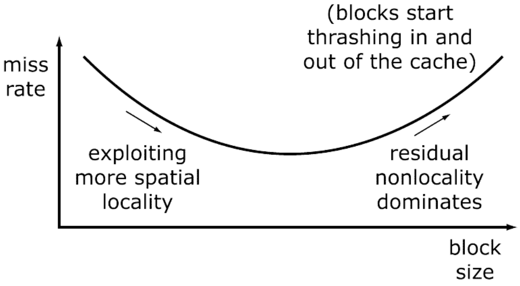
⇓ ability to tolerate spatial nonlocality ↓

Block Size and Misses [7.4.5.1.2]

Easy to see for the extreme case:



Block Size and Misses [7.4.5.1.3]



Associativity and Misses [7.4.5.2]

More

The miss rate decreases as associativity increases, but performance may or may not increase.

Associativity and Misses [7.4.5.2.1]

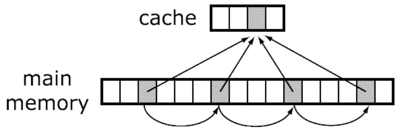
Many misses are caused by inflexible block placement in the cache, i.e., when blocks can go to only certain cache places. Recall

- direct mapped—block has only one place
- n-way set associative—block has n places
- fully associative—block can go anywhere

Blocks that map to the same locations in the cache are said to **interfere** or **conflict**.

Associativity and Misses [7.4.5.2.2]

Conflict can cause severe problems with certain array access patterns.

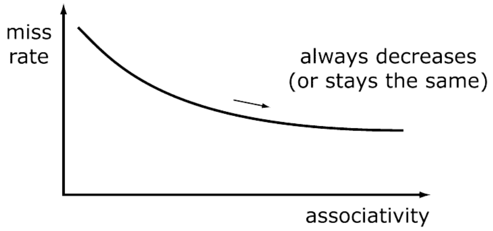


Interference can be reduced by increasing associativity:

↑ fully associative
set associative
direct mapped

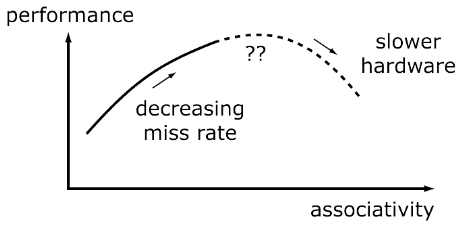
Associativity and Misses [7.4.5.2.3]

associativity ↑ ⇒ miss rate ↓



Associativity and Misses [7.4.5.2.4]

associativity ↑ ⇒ hit time ↑ ⇒ performance ↑↓?



Miss Classification [7.4.5.3]

More

To summarize miss behavior, we can organize misses into three categories: **compulsory**, **capacity**, and **conflict**.

Miss Classification [7.4.5.3.1]

Compulsory misses:

- These are misses resulting from the **first access** to a block.
- Also called **cold-start** misses

To reduce compulsory misses, **increase the block size** to exploit more spatial locality.

Miss Classification [7.4.5.3.2]

Capacity misses:

- These are misses caused by a **full cache**. When the cache is full, blocks may often be forced out and then reloaded.
- Note that these misses will occur even with a fully associative cache.

To reduce capacity misses, **increase the size of the cache**.

Miss Classification [7.4.5.3.3]

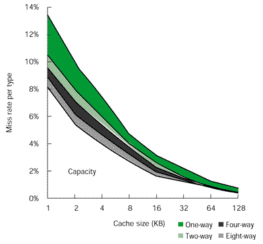
Conflict misses:

- These are misses caused by **inflexible block placement**.
- In direct mapped and set associative caches, blocks may be forced out and reloaded even though there are plenty of empty locations (i.e., not capacity misses).

To reduce conflict misses, **increase the cache's associativity**.

Miss Classification [7.4.5.3.4]

Figure 7.30, p.610



Reducing the Miss Penalty [7.4.5.4]

More

Even if misses are infrequent, a miss penalty of 50 to 100 clock cycles would significantly degrade performance.

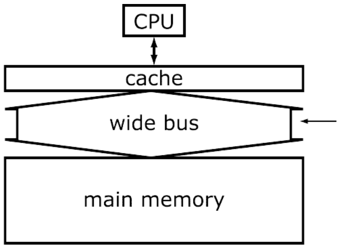
The miss penalty can be reduced using a variety of techniques.

Reducing the Miss Penalty [7.4.5.4.1]

- Use **intelligent transfers** when the needed word is in the middle of a block:
  - **early restart**—resume execution as soon as the word comes in; do the rest of the block transfer in the background
  - **requested word first**—start transfer with the needed word and "wrap around"
- Instead of stalling the pipeline waiting on a block, use **dynamic pipeline scheduling** to execute instructions out of order.

Reducing the Miss Penalty [7.4.5.4.2]

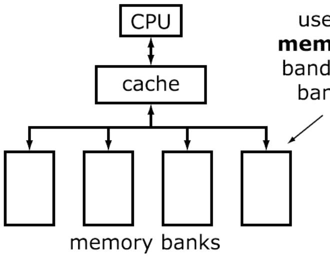
- Increase memory **bandwidth**



transfer more than one word at a time (128 bits, 256 bits, ...)

Reducing the Miss Penalty [7.4.5.4.3]

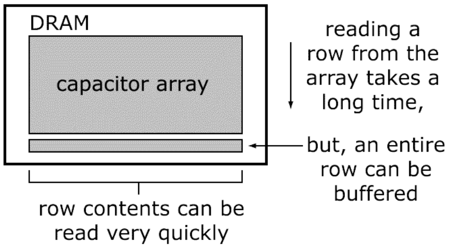
Take advantage of bus speed > memory speed:



use **interleaved memory** to increase bandwidth (multiple banks working in parallel)

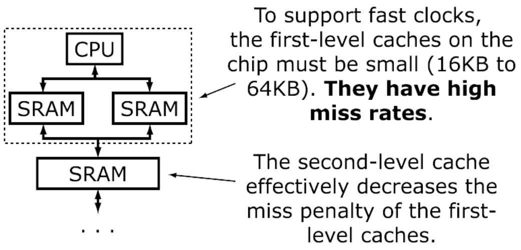
Reducing the Miss Penalty [7.4.5.4.4]

Take advantage of memory burst capability:


DRAM
capacitor array
reading a row from the array takes a long time,
but, an entire row can be buffered
row contents can be read very quickly

Reducing the Miss Penalty [7.4.5.4.5]

. Use **multilevel caches**



To support fast clocks, the first-level caches on the chip must be small (16KB to 64KB). **They have high miss rates**.

The second-level cache effectively decreases the miss penalty of the first-level caches.

Virtual Memory [7.5]

Virtual Memory Uses

Address Translation

Handling Page Faults

Virtual Caches

Virtual Memory Uses [7.5.1]

Overview of Uses

Increasing Effective Memory Size

Sharing Memory Space Efficiently

Controlling Memory Space Access

Overview of Uses [7.5.1.1]

More

Caches are used to **increase the effective speed** of main memory. Virtual memory is used to **increase the effective size** of main memory, but it also has other uses unrelated to the large and fast illusion of the memory hierarchy.

Overview of Uses [7.5.1.1.1]

Virtual memory has a variety of uses:

A. Increase the **effective size** of main memory
B. Allow simple, efficient **sharing** of main memory **among multiple programs**
C. Prevent multiple programs from **interfering** with each other (whether maliciously or accidentally)

Overview of Uses [7.5.1.1.2]

Note that uses B and C are **not** part of the large and fast illusion of the memory hierarchy. Why combine them with virtual memory?

Use A **requires an address translation** and this address translation is a convenient place to support B and C.

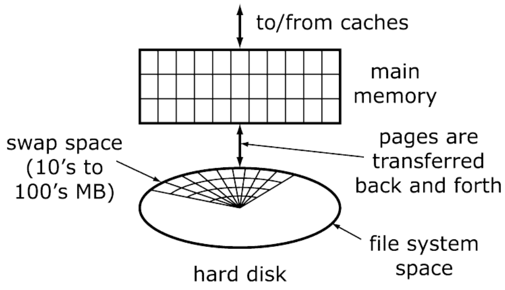
Supporting B is essentially free, and C only requires the addition of some control bits.

Increasing Effective Memory Size [7.5.1.2]

More

Virtual memory allows main memory to be used as a "cache" for a large **swap space** on a disk. Blocks are called **pages** and misses are called **page faults**.

Increasing Effective Memory Size [7.5.1.2.1]


to/from caches
main memory
swap space (10’s to 100’s MB)
pages are transferred back and forth
file system space
hard disk

Increasing Effective Memory Size [7.5.1.2.2]

Page sizes range from 4 KB to 64 KB.

The increase in effective memory size lets you

- Run larger executables with more data (not very effective for scientific codes)
- Keep multiple programs open in particular states
- Avoid file manipulation in software (CAD tools use this a lot)

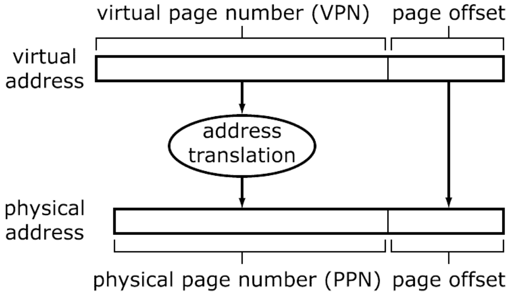
Increasing Effective Memory Size [7.5.1.2.3]

Since main memory acts like a cache, some mechanism must specify where a particular page is placed in main memory: **address translation**.

virtual address—the address generated by the program

physical address—the address actually used to access main memory

Increasing Effective Memory Size [7.5.1.2.4]



Increasing Effective Memory Size [7.5.1.2.5]

Virtual Memory	Caches
page offset	byte offset
VPN	block address
PPN (This "index" is found from a table instead of simply VPN bits.)	index

Note:
virtual address size: set by ISA
VPN, PPN, page offset sizes: set by hardware

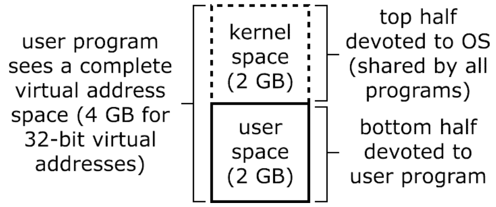
Sharing Memory Space Efficiently [7.5.1.3]

More

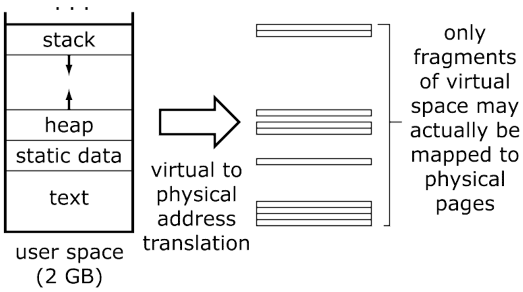
Different programs can have different address translations. Thus, **every program can "see" one large memory space with only it and the OS**.

Sharing Memory Space Efficiently [7.5.1.3.1]

Typical virtual memory arrangement:

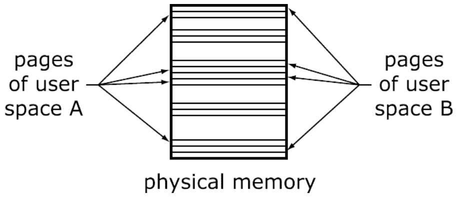

user program sees a complete virtual address space (4 GB for 32-bit virtual addresses)
kernel space (2 GB)
user space (2 GB)
top half devoted to OS (shared by all programs)
bottom half devoted to user program

Sharing Memory Space Efficiently [7.5.1.3.2]


. . .
stack
heap
static data
text
user space (2 GB)
virtual to physical address translation
only fragments of virtual space may actually be mapped to physical pages

Sharing Memory Space Efficiently [7.5.1.3.3]

User spaces from different programs can be easily mixed in physical memory:



Controlling Memory Space Access [7.5.1.4]

More

Virtual memory is the **foundation** for all computer security.

During the address translation, a memory access can be checked to see if the process has permission to read the requested area of memory.

Controlling Memory Space Access [7.5.1.4.1]

Memory restrictions are not useful if they can be changed by any process. Thus, a processor needs at least two modes:

- **kernel mode**—ability to access and change everything (used by the OS)
- **user mode**—restricted access; cannot change the virtual memory translations, etc.

On the MIPS, user programs can switch to kernel mode in a restricted way with "syscall".

Controlling Memory Space Access [7.5.1.4.2]

With different modes protecting the integrity of the address translation, we can set up access restrictions for a given process:

- Make some pages inaccessible in a mode
- Make some pages **read only** in a mode
- Give **full access** to some pages in a mode

Various combinations of access restrictions for the kernel and user modes are possible.

Address Translation [7.5.2]

Page Tables

Making Page Tables Smaller

Making Page Tables Faster

Page Tables [7.5.2.1]

More

The miss penalty for virtual memory is **extremely high**: on the order of 10^6 clock cycles.

For the absolute minimum miss rate, a fully associative mapping is used: any virtual page can be placed into any physical page.

Page Tables [7.5.2.1.1]

For the fully associative mapping, do we need lots of hardware comparators?

In virtual memory, the blocks (pages) are **large**, and there are far **fewer** of them than cache blocks.

Thus, it is feasible to have a lookup table with an entry for every VPN (block address). The table eliminates the comparator hardware but it takes time to do the lookup.

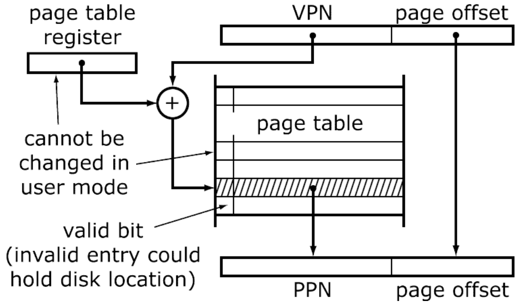
Page Tables [7.5.2.1.2]

The lookup table is called a **page table**. The address of a table entry is the VPN; the main contents of an entry is the PPN.

The user space of each process is mapped by a separate page table:

- This allows each process to have an uncluttered view of memory
- This prevents processes from interfering—a process cannot "escape" its page table.

Page Tables [7.5.2.1.3]



cannot be changed in user mode

valid bit (invalid entry could hold disk location)

Making Page Tables Smaller [7.5.2.2]

More

Even though page tables have a reasonable size for 32-bit addresses, we can divide the tables into **segments** to make more efficient use of memory.

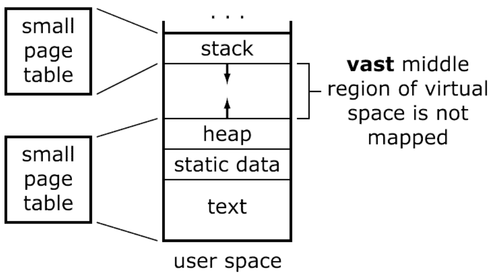
Making Page Tables Smaller [7.5.2.2.1]

If the VPN is 20 bits and each page table entry is 4 bytes, 4 MB are required for the page table of each program. **Imagine the problem with 64-bit addresses!**

Page table size can be reduced by not mapping certain areas of virtual space:

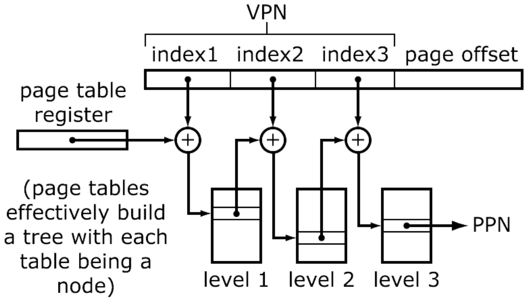
- Just map the top (stack) and bottom (heap)
- Or, support a generalized segment structure with a hierarchical table

Making Page Tables Smaller [7.5.2.2.2]



vast middle region of virtual space is not mapped

Making Page Tables Smaller [7.5.2.2.3]



(page tables effectively build a tree with each table being a node)

Making Page Tables Faster [7.5.2.3]

More

Address translation is required for **every** memory access; so, it must be **fast**. We cannot afford to do table lookups for every memory access. A special translation **cache** can be used to speed up this process.

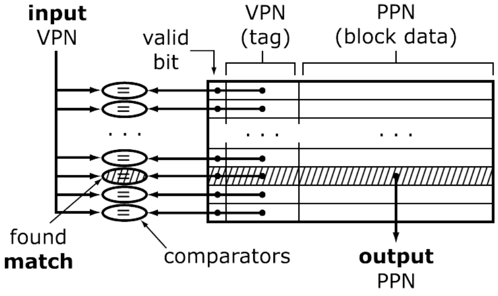
Making Page Tables Faster [7.5.2.3.1]

At any given time, there are usually only a few pages that are being actively accessed. The active pages are called the **working set**.

A small cache can hold the translations for the pages in the working set—the historical name for this cache is "**T**ranslation **L**ook-Aside **B**uffer" or **TLB**.

TLBs are usually small, built into the processor, and fully or set associative.

Making Page Tables Faster [7.5.2.3.2]



Making Page Tables Faster [7.5.2.3.3]

Watch out for the different conceptual levels of caching here:

- In virtual memory, the **main memory is a cache** for the swap space on the disk.
- The **page tables hold the mapping** that tells where a page on the disk is placed in main memory (for regular caches, the mapping is simple—extract a few bits from the block address)
- The **TLB is a cache for the page tables**.

Handling Page Faults [7.5.3]

More

Virtual memory **must** use a **write back** scheme since writing through is too slow even with a write buffer. (Thus, page tables need a dirty bit)

Handling Page Faults [7.5.3.1]

Page faults are handled by the OS. The miss penalty is enormous anyway due to the disk so there is plenty of time for software control.

Note that instructions must be **restartable**:

- On a page fault, the instruction must be stopped before it changes visible state.
- Resume execution when the page is ready.

A simple load/store ISA helps tremendously!

Handling Page Faults [7.5.3.2]

Virtual memory may use random or LRU replacement. One approximate LRU scheme is

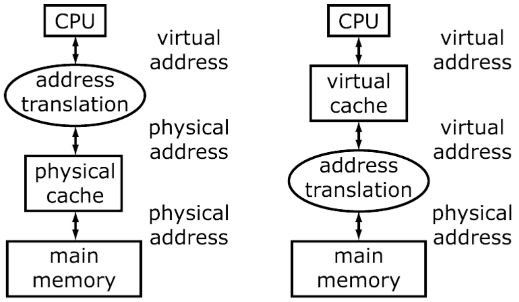
- Give each page table entry a **use bit** (in addition to the valid and dirty bits)
- Periodically **clear** all use bits
- When a page is accessed, **set** its use bit
- When choosing a page to **replace**, find one with a **cleared use bit** if possible (such a page would not have been accessed during the last time period)

Virtual Caches [7.5.4]

More

Where do we put our caches—before or after the address translation? This leads to the concept of **physical** and **virtual** caches.

Virtual Caches [7.5.4.1]



Virtual Caches [7.5.4.2]

	Virtual Cache	**Physical Cache**
Adv.	avoids translation delays	simple
Disadv.	must be flushed when processes switch aliasing	blocks must be flushed when a page is replaced

Virtual Caches [7.5.4.3]

Notes:

- Virtual cache flushing can be eliminated by appending the process ID to the virtual address (like the TLB).
- **Aliasing**—two processes share a physical page but it is represented in the cache at different virtual addresses—updating one cache block will not update the other
- You can **mix** virtual and physical, e.g., virtual first level and physical second level