Chapter

Cache

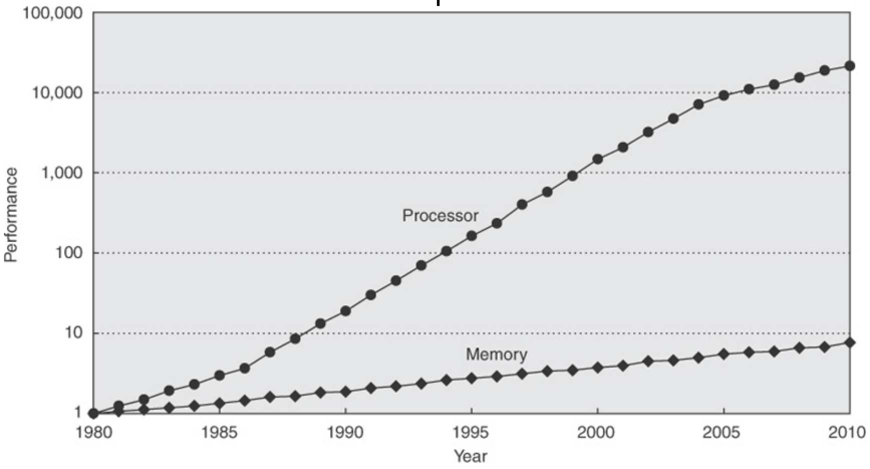
More and faster

Ideally one would desire an infinitely large memory capacity such that any particular word … would be immediately available …. We are …. forced to recognise the possibility of constructing a hierarchy of memories, each of which has a greater capacity then the preceding but which is less accessible.

A.W. Burks, H.H. Goldstine
A.Von Neumann
Preliminary Discussion of the logical design of an
Electronic Computing machine

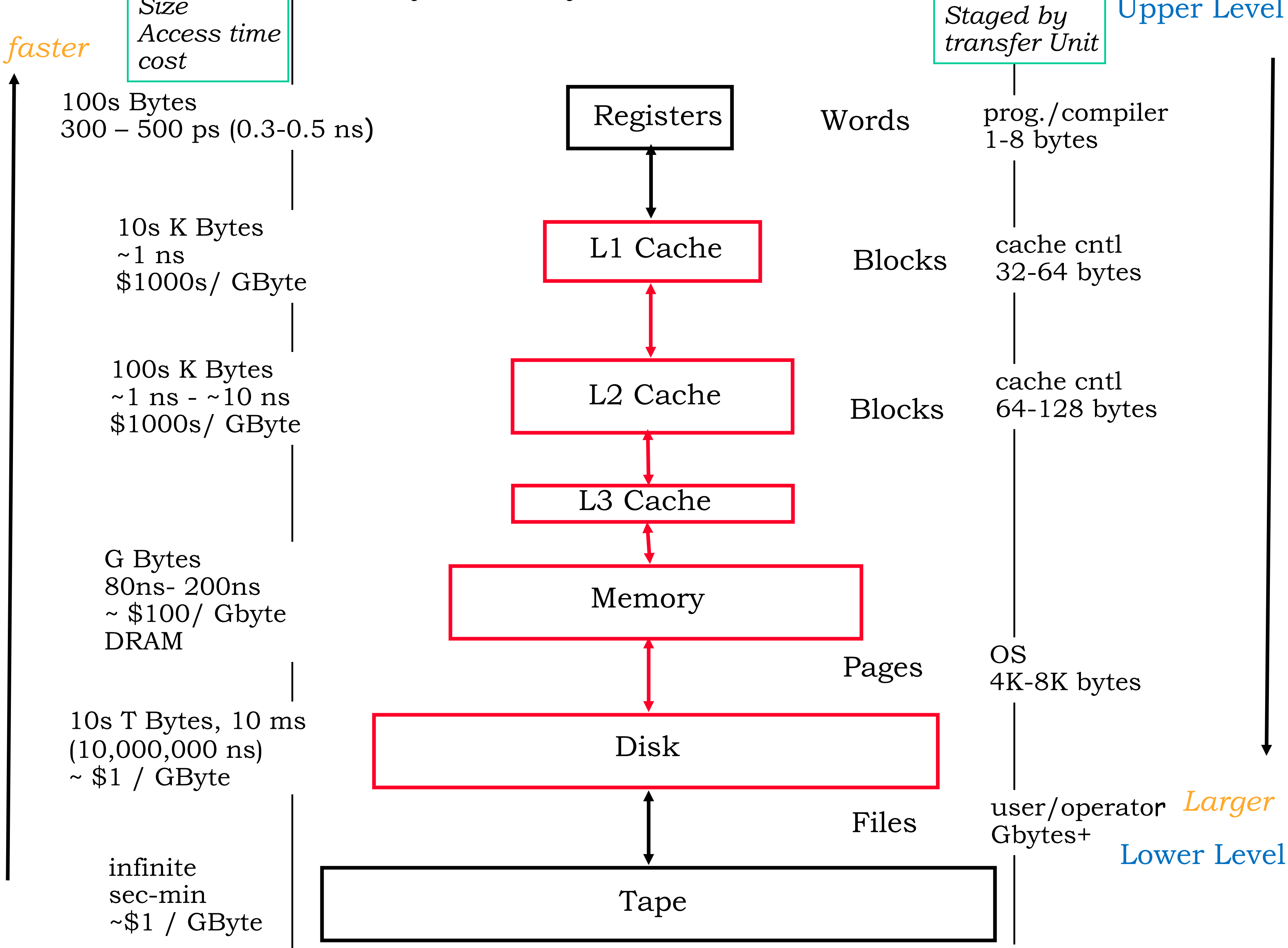
Baby 1948: Limited by memory access time.
Apple II (1977) CPU: 1000 ns; DRAM: 400 ns

Since 1980 memory speed times 10
CPU speeds times 20,000



© 2007 Elsevier, Inc. All rights reserved.

Memory hierarchy



Size
Access time
cost

faster

Staged by
transfer Unit

Upper Level

100s Bytes
300 – 500 ps (0.3-0.5 ns)

Registers

Words

prog./compiler
1-8 bytes

10s K Bytes
~1 ns
$1000s/ GByte

L1 Cache

Blocks

cache cntl
32-64 bytes

100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte

L2 Cache

Blocks

cache cntl
64-128 bytes

L3 Cache

G Bytes
80ns- 200ns
~ $100/ Gbyte
DRAM

Memory

Pages

OS
4K-8K bytes

10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte

Disk

Files

user/operator
Gbytes+

Larger

Lower Level

infinite
sec-min
~$1 / GByte

Tape

Accessing

Fast access

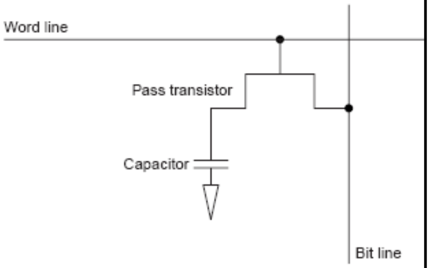
Aim computer architect: To provide sufficient memory at an economic price.

Fast memory tends to be expensive and volatile.
Static RAM (SRAM)
    0.5ns – 2.5ns, $2000 – $5000 per GB
Dynamic RAM (DRAM)
    50ns – 70ns, $20 – $75 per GB
Magnetic disk
    5ms – 20ms, $0.20 – $2 per GB
Magnetic Tape:
    Needs loading – sequential read.
    Seconds – minutes

Ideal memory: cheap, fast, reliable, permanent.

*Hierarchy :*Lots of cheap storage – decreasing amounts of more expensive memory. Move from cheap to more expensive as required.

DRAM



Readout on rising and falling clock edge.
DDR
Also how many bits per cycle. Data path width
DDR3 is 64bit

Implementation

Implementation details of RAM, help to explain why we use both DRAM and SRAM

DRAM is very simple – 1 transistor and 1 capacitor per bit.

Charged capacitor equals 1, uncharged 0.

Capacitor charge slowly leaks away.

Need to refresh (c.f. Baby). Read contents and write back.
Refresh done on chip.
Multi-megabytes time problem, structure to allow to refresh a row with a single read/write.
Refresh overhead small.

Memory access is stored in a square array.
Access is done by specifying row number and column number. Chip takes row and column through input chips. *(Pin count)*

Access time is in the 10s of nanosecond. It stays in step with SRAM performance but a factor 10 down.

ECC – Error correcting codes. On chip correction

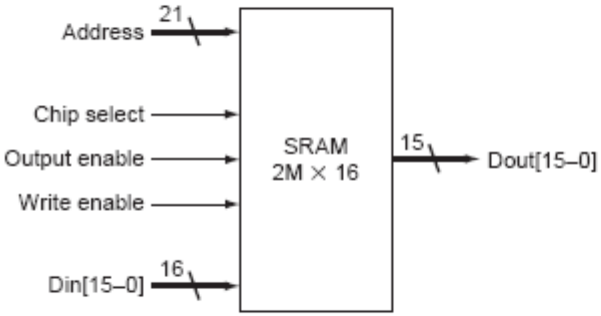
Simple structure – cheaper to make per cell.
Much smaller per cell.
Used for cache

Cache

SRAM

Implementation

SRAM access times are 1-5ns depending on the chip memory size.



SRAM/DRAM cache, lots of development

Sufficient address lines to address all memory. Memory smaller and used for main memory – so speed is vital.

Needs ~6 transistors per cell and so is more expensive and takes up more space than DRAM & more power

Drive the clock too fast and memory becomes unreliable

Why not run just SRAM? At almost any price point a combination of DRAM and SRAM will give better performance.

Low power SRAM typically same speed as DRAM

Again minimum times are given by minimum setup times, write enable is a not edge triggered. Pulse with minimum width for stable operation.

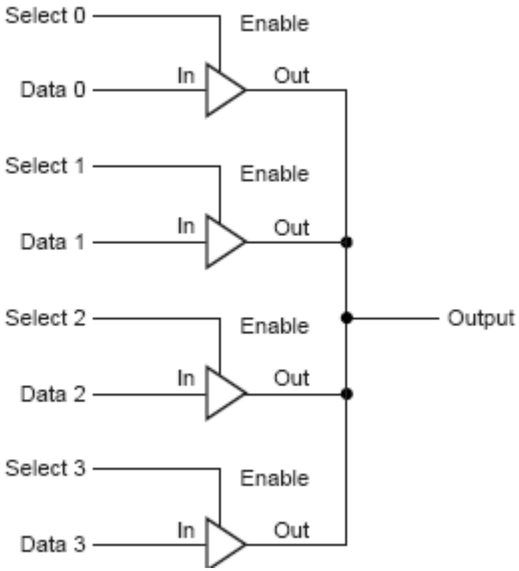
Output via shared line

Cache

SRAM

SRAM/DRAM cache,
lots of development

Tri-state buffer



The data line is selected by a signal on the select line.
Once selected the Data line can be high (1)
or low (0)

If not selected. High impedance mode – one can be read without the others interfering.

Modern RAM supports **burst mode**.
Provide starting address and length.
After setup sequential addresses are transferred once each clock cycle.

Drive the clock too fast and memory becomes unreliable

Cache

Errors

Number of 1's is even or odd.

Error Correcting codes ECC

Memory bits can end up in the wrong state.

Cosmic rays can cause a bit to flip.
Higher memory density → smaller cells → less charge to flip a bit. More errors.

ECC memory has extra information to detect bad data.
Simplest is a parity bit. 1 per 8 bits. Makes the parity sum odd (or even).

Any single bit error can be detected (but not corrected). Flipping two bits produces a valid word.

Error correcting codes are more complex.
They add bits and define only certain bit patterns as being valid
.
In particular single errors produce codes which are only 1 flip from the correct sequence, but two flips from any other valid sequence.

Most common are Hamming codes

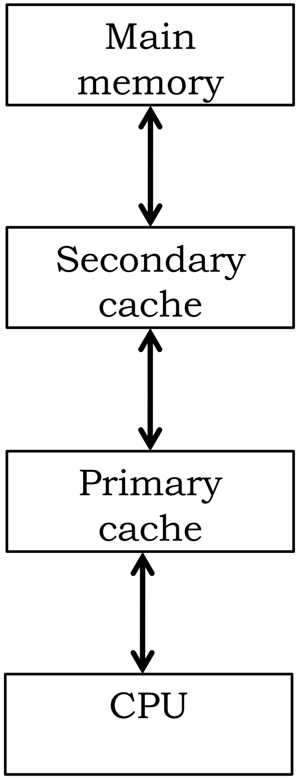
Memory controllers can implement forward error correction, where the bad data is corrected before the data is transferred without referring back to memory

Error detecting

Language usually allows us to detect and sometimes recover from single character errors

Computer
computer
commuter

Cache



Overview

Fetch from main memory is slow.
Fast memory is expensive.

Solution a hierarchy which has large amounts of cheap memory and closer to the processor, one or two stages of progressively faster memory *the cache*

But data still has to flow all the way down from main memory to the CPU. How does this save time?

Spatial locality
Temporal locality

How does the CPU/MMU know where to look in the cache for addresses which locate the data/instruction in main memory?

Locality

Locality

Move from cheap to more expensive as required … if data is accessed only once or completely at random this would have little benefit.

In general programs memory access is rather highly constrained by the (observed) principle of locality.

Temporal locality

Items tend to be accessed repeatedly

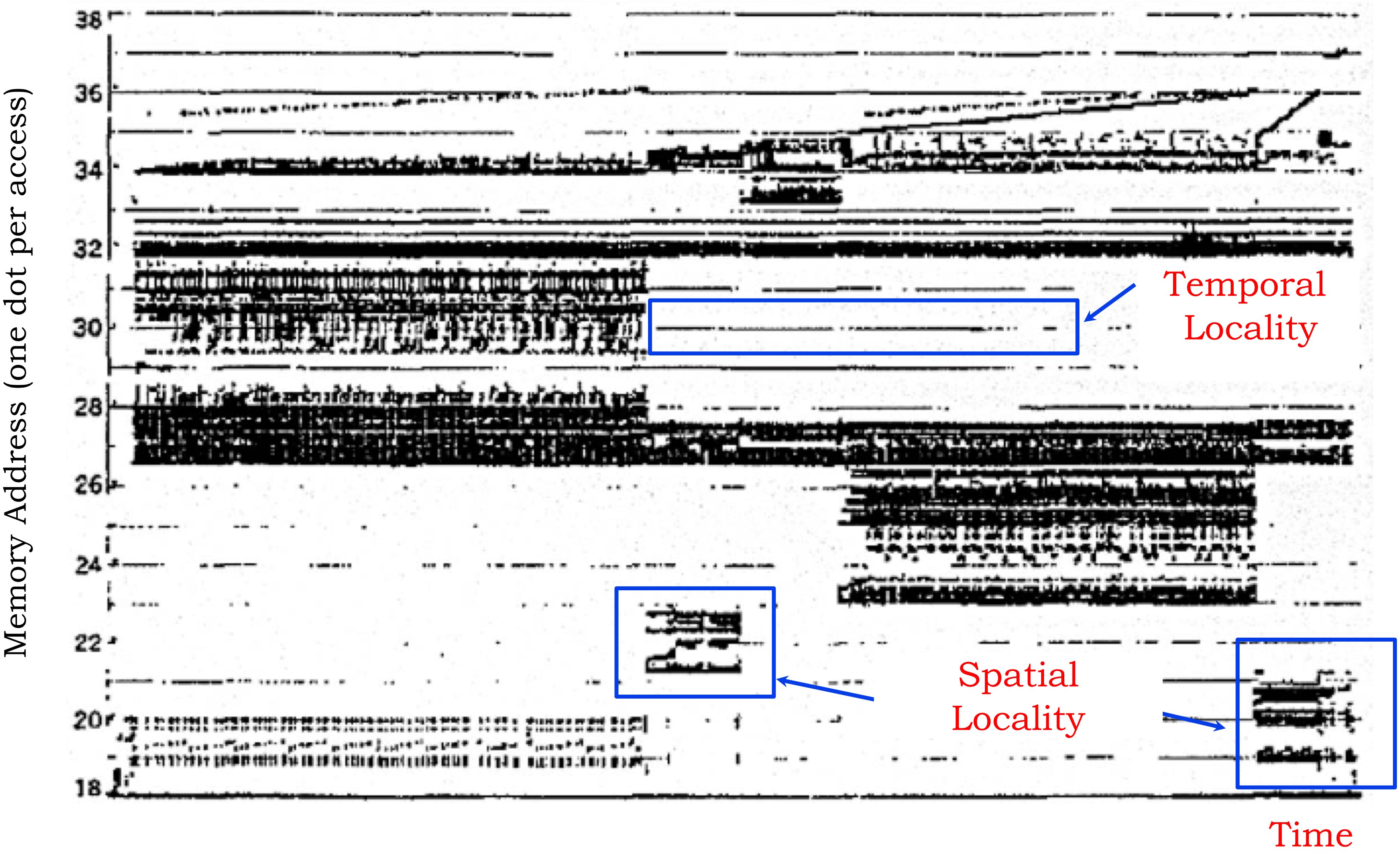
During execution of a loop

Spatial locality

If the next item is not the same as the last it is likely to be nearby.
Code tends to be executed sequentially.
Data items in arrays are arranged sequentially.

Memory accesses as a function of time

Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)



Temporal Locality

Spatial Locality

Already there

Pre-fetch

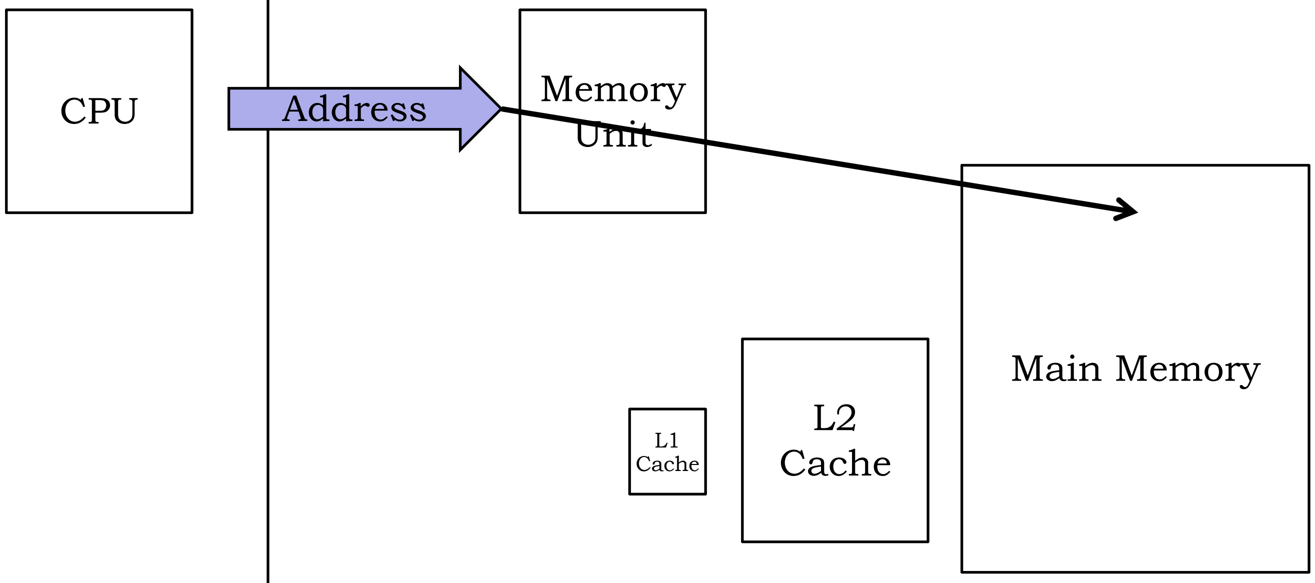
Fetch item/instruction, plus surrounding items and they are likely to be useful.

Fetch done by independent hardware

Time wasted when not used, but only the time to fetch the real instructions. Pre-fetching unused instructions has no explicit overhead.

How does the Memory controller know where to find things?
What happens when a variable in cache is given a new value?

Items already in fast storage when required



Cache

Cache performance

When the CPU wants something already in cache that is a *hit*. If it is not in cache that is a *miss.*

Cache miss rate is a measure of how well the system is doing.

Miss rate (per memory access) = $\dfrac{\text{Misses}}{\text{Hits + Misses}}$

A Number of possible measures

May also refer to
Misses per instruction = Miss Rate x $\dfrac{\text{Accesses}}{\text{Instruction}}$

But how bad is a miss?

<memory access time>
= (Hit time) x(hit rate) + (Miss rate)*(Miss Penalty)

Where Miss penalty is the time to retrieve the item from further up the tree.

<memory access time> is still not as good a measure as execution time, but is useful for feature comparisons.

Cache Actions

On cache hit, CPU proceeds normally

On cache miss
Stall the CPU pipeline
Fetch block from next level of hierarchy

Instruction cache miss
Restart instruction fetch

Data cache miss
Complete data access

Cache can be multi-level. Up to three cache levels are available on some modern systems.
Transfers between adjacent levels.
Consider only two levels at a time.

Miss means that data is copied from the next level down into this level of cache.

A cache miss at one level down can only be triggered by the level above cache or CPU.

Levels normally expense and speed of technology.
At constant “technology speed” larger caches are slower. May split just for speed.

Cache Effect

Improve CPU
Misses become more important.

Decrease base CPI: larger proportion on stalls

Increase Clock more cycles for a memory stall

Execution performance

A cache miss, means the CPU needs to wait a certain number of cycles.

Miss Penalty = Access time + transfer time

Execution Time
= (Instructions + (Memory Stall Cycles))*period

$$= (\text{Instr Count})*\frac{\text{Accesses}}{\text{Instruction}} \text{ x (miss rate) x (penalty)}$$

Approximation – miss penalty is different for reads and writes.

Take an average ratio of reads/writes
    Differs between programs.
Read rate and penalty, write rate and penalty.
    Added complexity.

Increase transfer size.
May reduce miss rate, will increase miss penalty

Example

Pentium 4

Level 1 (D-Cache): Capacity=16K, Access=4 cycles

Level 2 (D-Cache): Capacity=1024, Access =18cycles

Main memory (D-Cache) Access = 180cycles

Level 1 is longer than one would want in a MIPS machine.

Caching terminology

Block (line): Unit of storage in the cache
Memory is divided into blocks – which map into cache locations.

Data Referenced:
Hit: Data in cache
Miss: Data not in cache – fetch it from higher level
May mean replacing something in cache

Design considerations:
Placement: where to place a block in cache
Replacement: what to remove (overwrite)
Granularity: block size, uniformity
Writes: action when value in cache is written to
Instructions & Data: Uniform or split cache

Miss types

Sources of misses

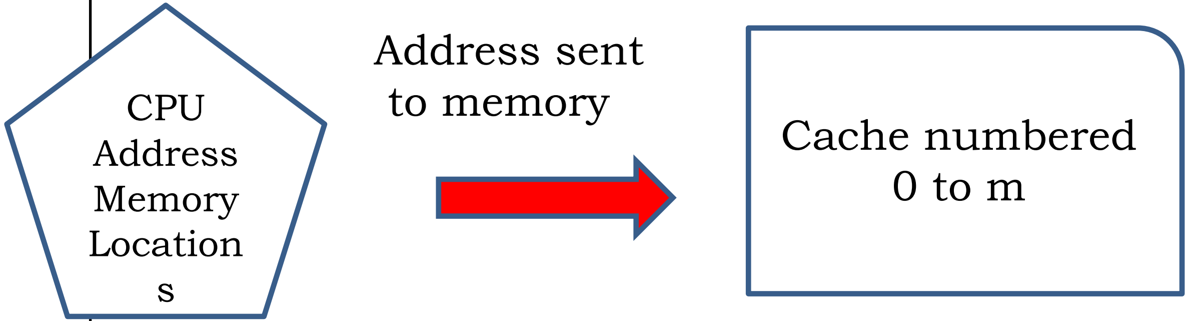
Compulsory: First time access and the block cannot be in the cache.

Capacity: If the programme needs more memory than is present in the cache, then some blocks will be discarded and later retrieved,

Conflict: If the cache organisation means that two blocks of main memory are written to the same area of cache. One might over write the other **even** while there is room in the cache.

Coherency: requirement to keep multiple caches consistent.

A Number of possible measures



CPU Address Memory Locations

Address sent to memory

Cache numbered 0 to m

Where is a block in put the cache?

How is a block found in a cache?

If the cache is full and there is a cache miss which block in the cache should be overwritten?

What happens when the cpu wishes to write a value into a memory location. Either in the cache or not in the cache

Memory locations 0 to N

Cache

Placement

Mapping from memory to cache

Store everything on **disk** non-volatile

Disks performances.
Spin speed – latency.
Areal density
RAID – multiple reads
SSD’s – energy rather than speed.

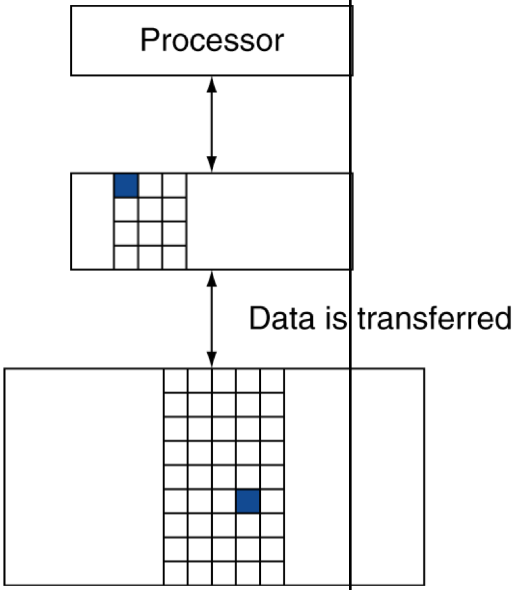
Fetch items (and nearby) items from disk to smaller **DRAM** memory

Main memory – different speeds/cost

Fetch items (and nearby) items from DRAM to smaller **SRAM** memory

Cache memory attached to CPU

When it comes to moving data into the cache – where does it go ?



The fact disk is non-volatile is a reason for using it

miss
miss ratio
miss
miss penalty

miss ratio + hit ratio = 1

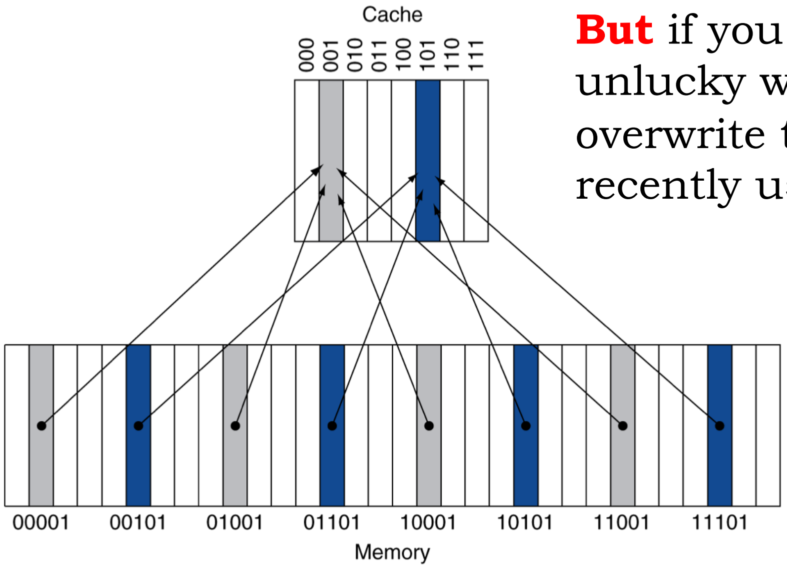
Direct

Direct mapped: no choice

Location determined by programme address (Block address) modulo (#Blocks in cache)

If the number of locations in cache is a power of just Use low-order address bits

Pretty algorithm



But if you are unlucky will overwrite the most recently used block

How do we know which data in 7a5c of the cache is being stored. Store the address (just high order bits)

Tag:

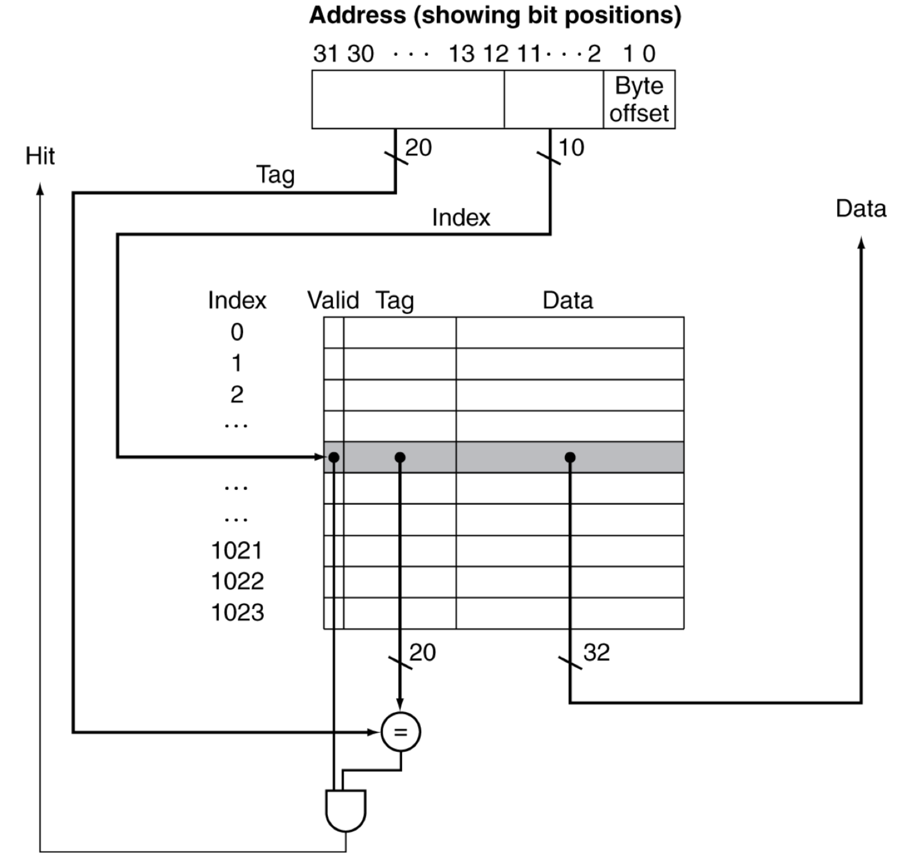
At the start nothing in cache … for each *block* we have a **valid** bit = 0 data not present

bit = 1 data present

Cache

Cache Addressing

Everything from the address



Address = 4681924 = 004770c4

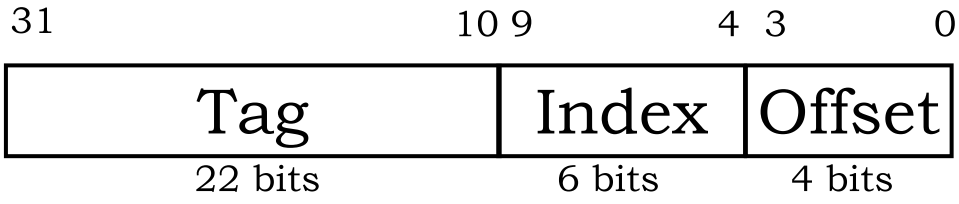
0000 0000 0100 1110 1110 00 001100 0100

Block address = 4681924/16 = 004770c

Block index = address mod 64 = 0c

Offset is 4 = 0100

Tag is 004770 …. Last digit is only 2 bits



31 10 9 4 3 0

Tag | Index | Offset

22 bits | 6 bits | 4 bits

64 blocks
16 bytes/block

Index to identify the block. Tag to check it is correct

Cache

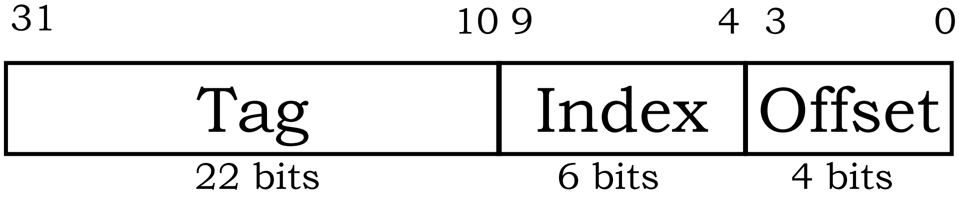
Deconstructing the address

Index: 6 bits that means there are 64 blocks in cache.

Offset: 4 bits which mean every block is 16 bytes long (could be words)

The TAG is the rest of the rest of the address.

For a given value of index and offset – any value of the tag corresponds to a position in memory which will be stored at that location of the cache.



31 10 9 4 3 0

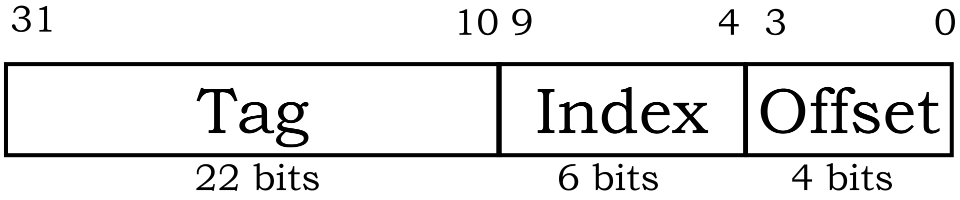
Tag Index Offset

22 bits 6 bits 4 bits

Cache Size

Total size for a cache

This showed blocks being 1 word – normally they are bigger – transfer extra data to utilise locality. Not too many or transfer time increases and hence miss penalty.
So each block has K words. K is a power of two for addressing. K words = 2^2*K bytes = 2^(k+2)
The cache has 2^m blocks – again a power of 2.
Block is then 2*(m+k) words or 2^(m+k+2) bytes.
32 bit address space – 2^32 bytes – 2^30 words
The cache is overloaded by 2^32/2^(m+k+2)
So the Tag must have 2^(32-m-k-2) bits -which come from the high order of the address.
The byte offset inside the block is the first k bits of the address – and the intervening bits give the block address inside the cache.
Total cache size then needs to include in addition to the data/instruction word the tag bits and the valid bit
.


31 10 9 4 3 0

Tag | Index | Offset

22 bits 6 bits 4 bits

64 blocks: 4 words
16 bytes/block
Overload is
2^(30-8) = 2^32

Cache

Cache Size

Cache Optimisation

Spatial locality indicates we should transfer large blocks of data.

Large cache is clearly good, **but** if size is fixed.

Large blocks mean fewer blocks:
    may overwrite block before it is finished with *temporal locality*
May lead to a higher miss rate;
    more to be transferred so larger miss penalty

Cache too large and we will see degradation

Cache hit/miss

Hit … access data

Miss

Need longer to fetch so stall the pipeline
Fetch required data from the next level
Miss Type:
Instruction --- restart instruction fetch
Data --- Complete data access

Cache

Anywhere

Fully associative: Unlimited choice

The block can be stored anywhere

Set associative: Limited choice

The block can be stored in a number of locations
A ***set*** is a group of blocks in cache.

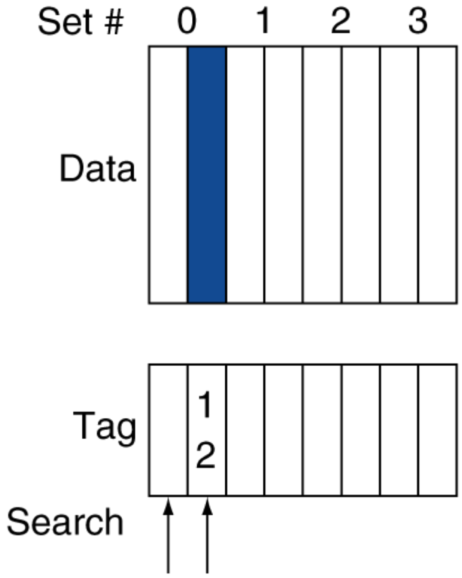
A block is mapped to the set, similar to direct mapped.
In this case set is chosen by

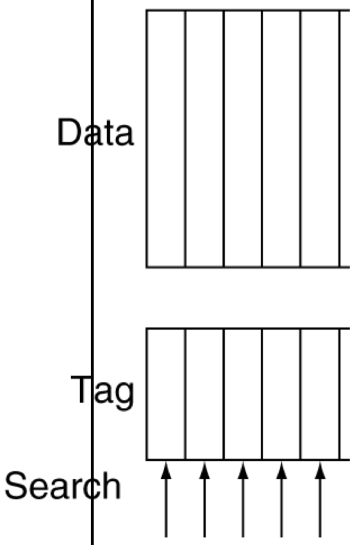
Block address) modulo (#Sets in cache)

compare
(Block address)
modulo (#Blocks in
cache)

Here each set has two blocks so the data can be placed in either block.
Two way associative

Also 4-way, 8-way, …. Fully.





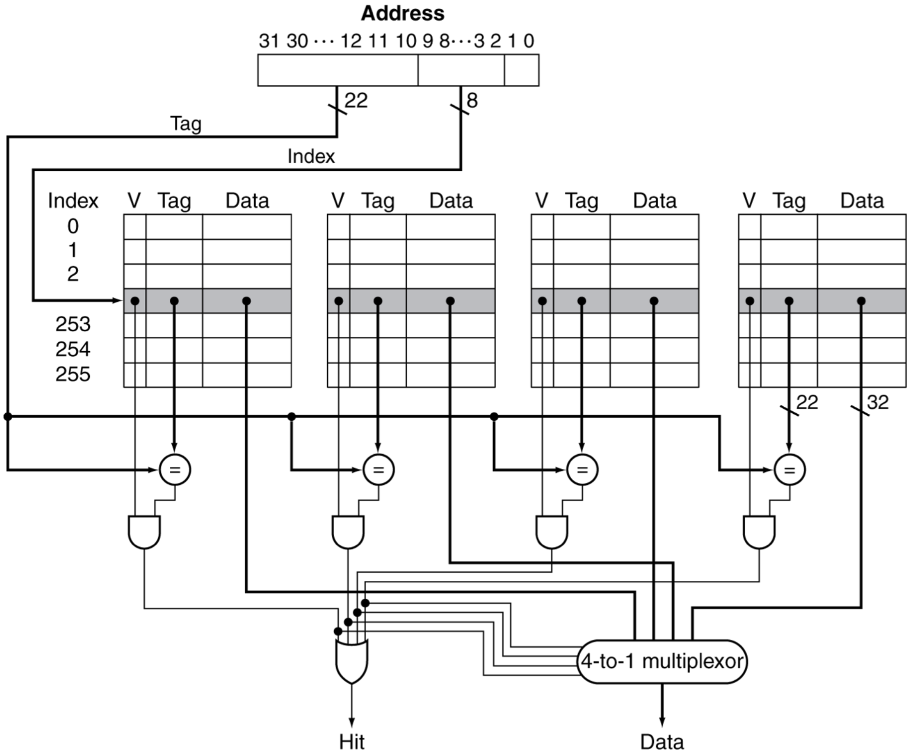
Cache

Improvements with associativity

Increased associativity decreases miss rate
But with diminishing returns
Simulation of a system with 64KB
D-cache, 16-word blocks, SPEC2000
1-way: 10.3%
2-way: 8.6%
4-way: 8.3%
8-way: 8.1%

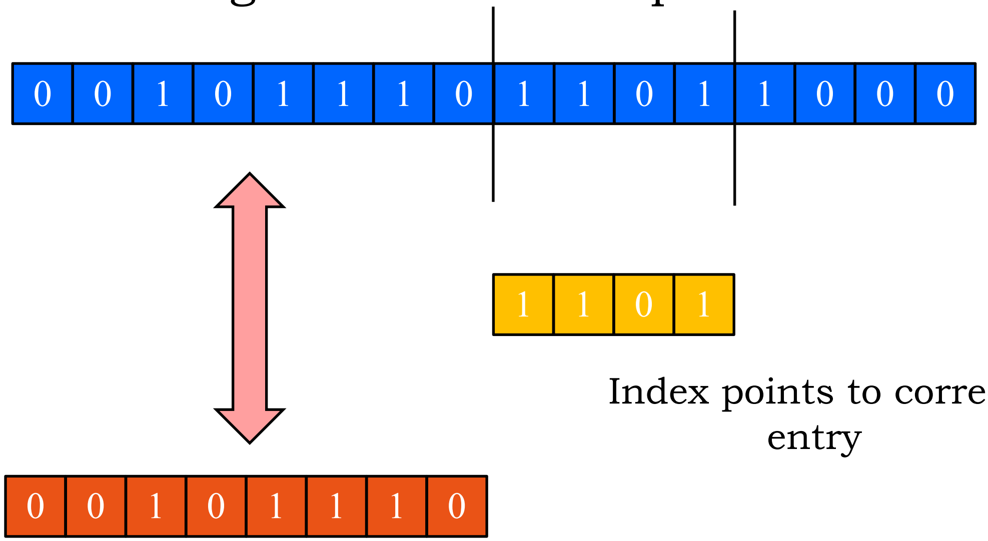
And increased complexity

Be careful when comparing systems.
Block size improves hit rate
Associativity improves hit rate
Total size has the largest performance.
Comparisons must be on same size memory, not same number of blocks



Considerations

Single associative means you get conflict misses, but checking the cache is simple



Index points to correct entry

Simple bitwise comparison to say if the access is a hit.

Fully associative means there are never conflict misses. So saving bandwidth.
But looking for a hit either means 1 cycle for every index position in the cache, thus increasing the hit time – or multiple comparisons.

Reduce miss rate, but increase hit time

Cache is not about reducing miss rate – but reducing average memory access time.

Memory address

Cache

Block replacement

Replacement algorithm

Cache is full, which block to throw out?

Direct mapped No choice. Simplest hardware.

Fully associative:
Random: easiest to implement. (Can use pseudo-random replacement, so it is predictable and makes testing easier)

Most common

Least-recently used (LRU): exploiting temporal locality. Need to store access time.

First in First out (FIFO) :
Easier to implement that LRU.

Two blocks, LRU is just alternate. Finish a block. Get another address, go to the other block, either the address is there or write it there.

LRU becomes harder as size increases.

For a large cache there is no difference. For a small cache LRU is better, but not by much.
Rate ~ 11%
Difference < 0.5%

Block
replacement (i)

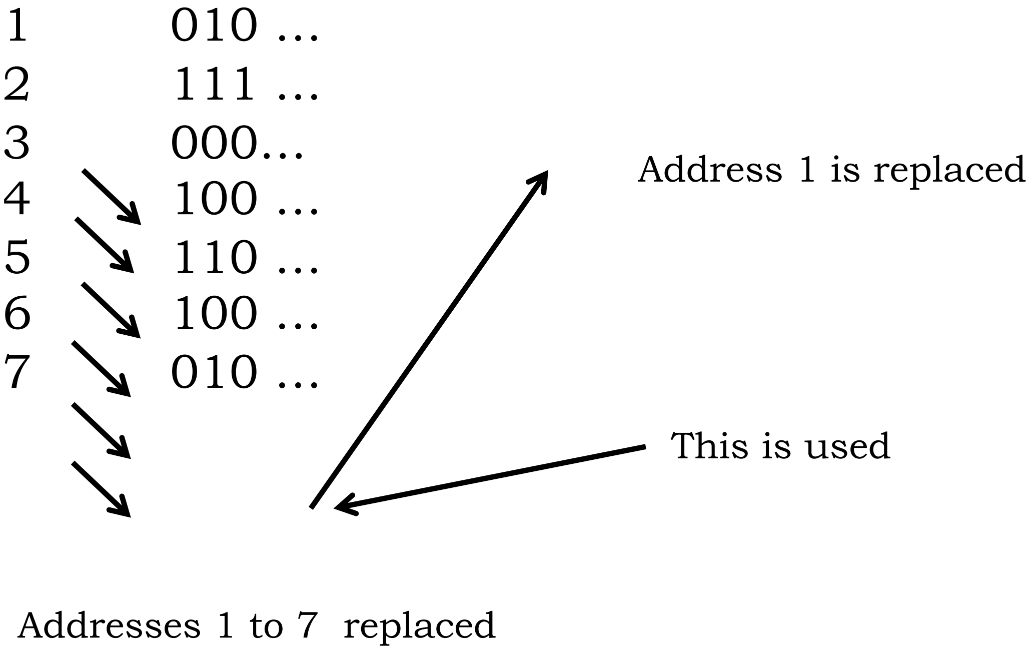
LRU

Seems like this must be the best – *locality*

But this means that you must keep the time for all accesses.
Further that you must be able to identify the least recently used block

Clearly checking is time consuming
n comparisons

Or keep a list associated with the time of last use



1	010 …
2	111 …
3	000…
4	100 …
5	110 …
6	100 …
7	010 …

Address 1 is replaced

This is used

Addresses 1 to 7 replaced

Block replacement (ii)

LRU

Or rewrite numbers

3	010 …	4	010 …
6	111 …	7	111…
2	000…	3	000..
1	100 …	2	100..
7	110 …	1	110 ..
1	100 …	6	100 ..
4	010 …	5	010 ..

You still need to overwrite a number of locations (increment mostly)
And now when you are throwing one out you have to search for the largest number. Miss penalty getting larger.

One might look to throw out blocks whose contents have not been changed – but that depends on the strategy for updating values in cache that are changed.

Block replacement

Replacement algorithm

Assoc:	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Least recently used is intellectually appealing **but** it is more complicated to implement and makes very little difference.

Increasing associativity has a small effect, cache size dominates.

64 to 256 is rather small and there will be a time penalty for the increased size

Approximations

NMRU

Not most recently used – just needs 1 bit

Victim – Next Victim

2 blocks status tracked in each associativity set

V (victim) – NV (next victim) – others are O(rdinary)

On cache hit

Promote NV to V

Promote an O at random to NV

Return V to 0

On cache miss

Replace V

Promote NV to V

Promote an O at random to NV

NMRU – means you are guaranteed not to kick out the last used block and V/NV means that you won’t kick out either of the last two.

Locality means that correlations are only short range.

Cache miss

Actions on a cache miss

A dedicated controller which:

Performs the memory access and fills the cache

Creates a stall (cf pipeline stall), not an interrupt.

Stall whole processor – easier than pipeline stall, but much more costly for performance.

After a miss the instruction register does not contain a valid instruction. So:

1.Set the PC to PC -4; points at instruction which generated the miss

2.Request read from memory and wait for completion (many cycles)

3.Write the data to cache; writing the upper portion of address (from ALU) to the tag field, set the valid bit

4.Restart execution.

Instruction is resent and this time results in a hit.

Interrupts requiree register store

Writing

cache example

A dedicated controller which:

Performs the memory access and fills the cache

	Write-Through	Write-Back
Policy	Data written to cache also written to lower level memory	Write data only to the cache. Update lower level when block falls out of the cache
Debug	Easy	Hard
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Cache

Cache writing

Assume all other instructions are single cycle
Effective CPI
= 0.88*1 + 0.12 × 100
= 12.9
Significant slow down

Cache Hit

Write through cache

Options

Update the cache **But** memory inconsistent

Could update memory as well.
But time consuming

Base CPI is 1, then write to memory is around 100 cycles 12% of instructions are stores

SpecInt92 benchmarks on Intel x86

Solution:
write buffer: holds data to be written to memory
CPU has no need to wait
stalls on write only occur if write buffer is full

Write back

If there is a write hit, only update the cache
Keep track of *dirty* blocks

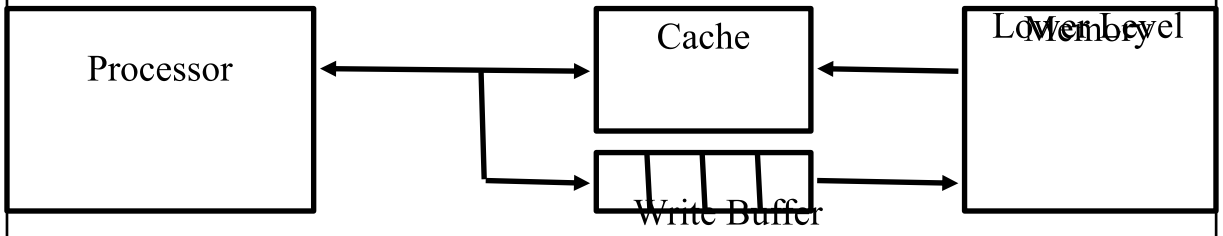
When a dirty block is overwritten
Write it back to memory

Cache miss

Allocate on miss: read the block into cache
Write around: don't fetch the block.

Write buffers

Cache write



SpecInt92 benchmarks
on Intel x86

CPU doesn’t stall on a write.
Write not just the register, but a number of writes
more localisation. Writes tend to come in bursts.

Creates potential RAW problems.

Spec2000 10% are stores. Store is 100 cycles.

100 instructions which are 100 cycles without store.
Becomes 90 (no store) + 10*100 (memory access) + 10(store execution) = 1100 cycles.

Loose a factor of 11!

Replacement strategies .1% - means .1 instruction or 10 cycles per 100 cycles or 10% slow down.
Not quite as insignificant as first appears.
.

Split Cache

Separate Instructions and Data (eg MIPS)

I-Cache **D-Cache**

Different requirements
Eg no write-back
Can access both instruction and data simultaneously

I-Cache miss rate 0.4%
D-Cache miss rate 11.4%

Spec2000 benchmarks on Embedded MIPS

Main Memory

Performance

Main memory is DRAMs. Size not speed
Bus clock from memory to cache is slower than CPU clock.

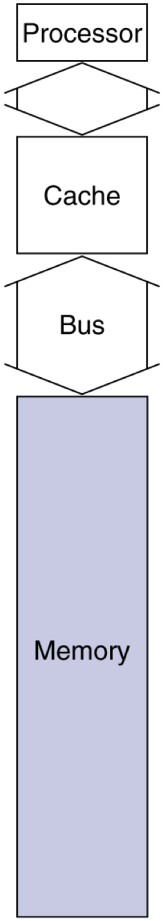
1 cycle to transfer address
15 cycles per DRAM access
1 cycle per word transfer

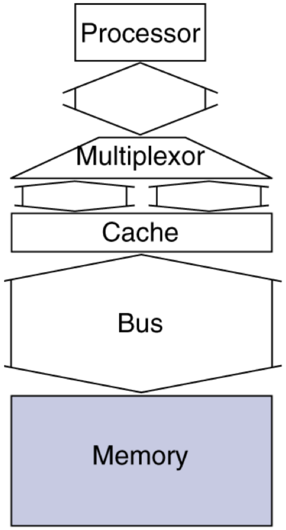
Larger block bigger penalty.

For 4 word cache block

Cycles are 1+4*15+4*1 = 65 cycles
Miss penalty

16/65 bytes/cycle = 0.25





Increase bus width to 4 words
Cycles = 1+15+1=17

So width of the memory to cache bus has a dramatic effect on the cache penalty
bytes/cycle = 0.48

Remember this is only one part of the performance.

Cache

Memory organisation

Year	Size	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50

Source: Patterson Computer Organisation and DEsign

Higher Performance

Four bank interleaved – no increase in data path

Cycles are 1+15+4*1 = 20 cycles

Other possible improvements

Burst mode: only need full access time for first word. Subsequent words faster
DDR: transfer on the rising and falling clock edges. (Double Data Rate)
QDR: separate input and output on DDR

Quantitative
Need not transfer rate memory to cache, but on overall performance.

Memory stall cycles from cache misses =

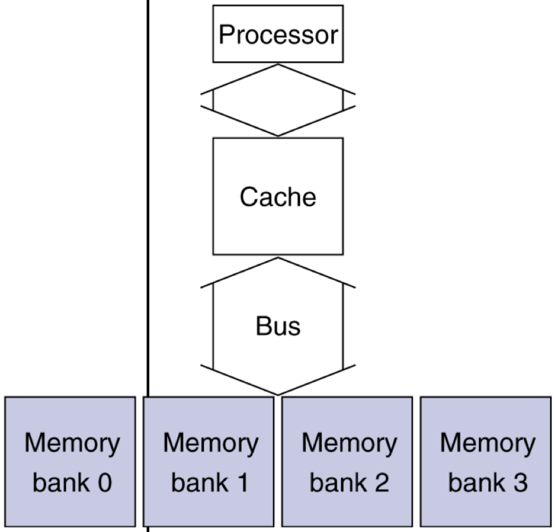
Memory accesses/program * Miss rate * Miss penalty
equivalent
Instructions/program * Misses/instructions * penalty

Average memory access time (AMAT)
= Hit time*Hit rate + Miss rate × Miss penalty

Eg hit time is 1 miss penalty is 20. Miss rate is x
What rate doubles the memory access time?

$$1*(1-x) + 20x = 2$$

$$19x = 1 \qquad x = 0.052$$



Processor

Cache

Bus

Memory bank 0

Memory bank 1

Memory bank 2

Memory bank 3

DDR transfers on rising edge and falling edge

QDR separate inputs and outputs.

Cache

Effect on Performance

When CPU performance increased
Miss penalty becomes more significant

Decreasing base CPI
Greater proportion of time spent on memory stalls

Increasing clock rate
Memory stalls account for more CPU cycles

Can’t neglect cache behavior when evaluating system performance.

So in all improvements in processor performance will not translate into execution time improvements unless the cache performance keeps pace with the CPU

Optimisation

Cache optimisation

How can we distribute our resources in the cache so as to improve system performance.

<average access time> =

(Hit time)*(Hit rate) + (Miss time)*(Miss rate)

Rate is a fraction

Hennessey has a simplified version

1.Reduce miss rate	r
2.Reduce miss penalty	T
3.Reduce hit time	t

$$\langle t \rangle = t.(1-r) + T.r$$

$$= t - rt + Tr$$

If $t(1-r) = Tr$ equal contributions

$t - tr = Tr$ -> $1 - r = r.T/t$

$1/r - 1 = T/t$ -> $1/r = T/t + 1$

$r = t/(t+T)$ for $t \ll T$

$r \sim t/T$

$r > t/T$. Reduce r or t

$r < t/T$ reduce T

Cache