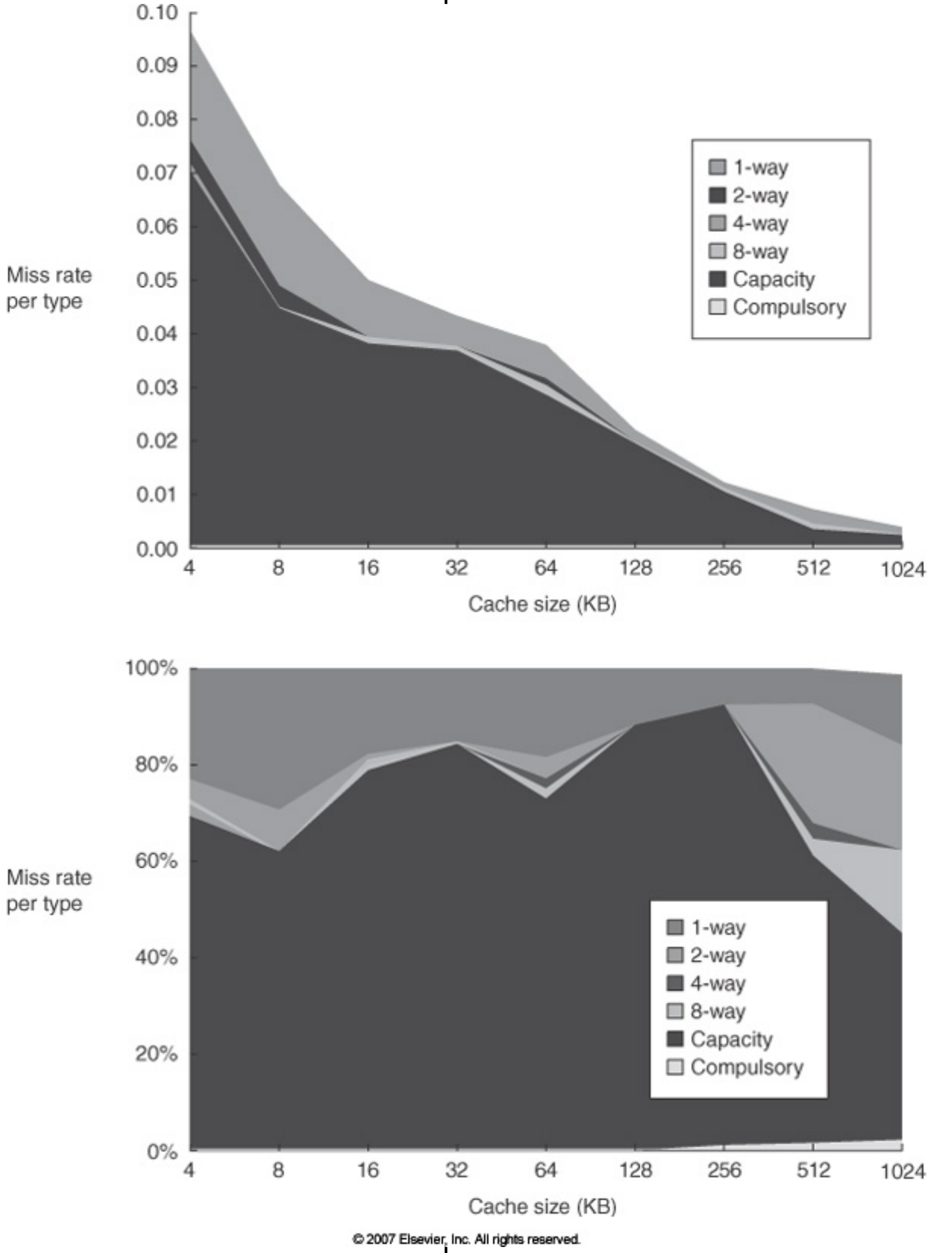
Cache Optimisation

“sometime he thought that there must be a better way”

Cache Optimisation

1. Reduce miss rate
   a) Increase block size
   b) Increase cache size
   c) Higher associativity
   d) compiler optimisation
   e) Parallelism
   f) prefetching (hardware and compiler)
2. Reduce miss penalty:
   a) Multilevel caches
   b) Write through cache
   c) critical word first
   d) merging write buffers
   e) Parallelism
   f) prefetching (hardware and compiler)
   g) increase cache bandwidth (pipelined, multi-banked, non blocking) caches
3. Reduce hit time:
   a) small simple caches
   b) way prediction
   c) trace caches

Which c is responsible?



Compulsory : Small

Capacity : Major

Conflict:

Four-way associate, is the extra misses going from 8 way to 4 way associate

From Patterson

Increase block size

At a given cache size, increasing block size firstly reduces miss rate.

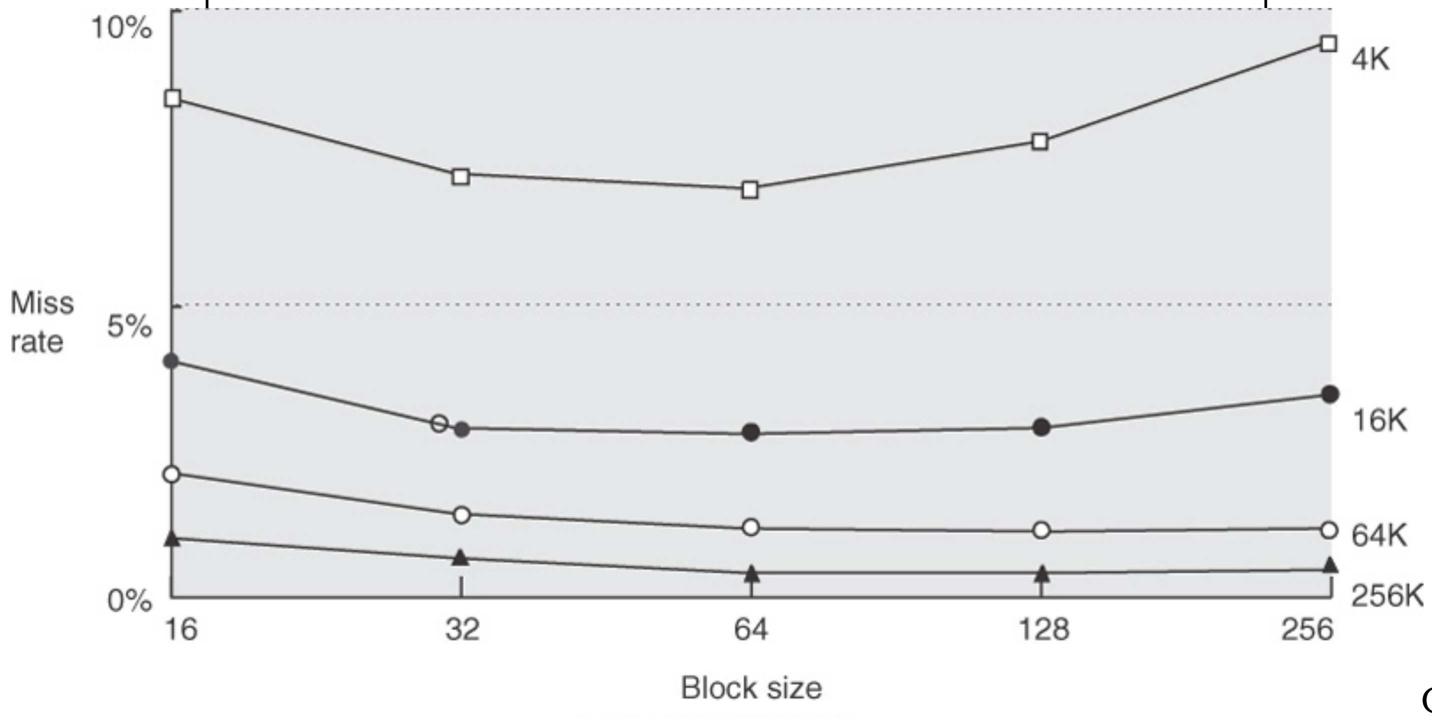
Compulsory misses drop because of spatial locality.

Reduce the number of blocks in a cache. Increase conflict misses and capacity misses.

Increases miss penalty (more data to transfer).

Penalty = latency + bandwidth.

The optimum is a balance.



Technique 2

Larger caches

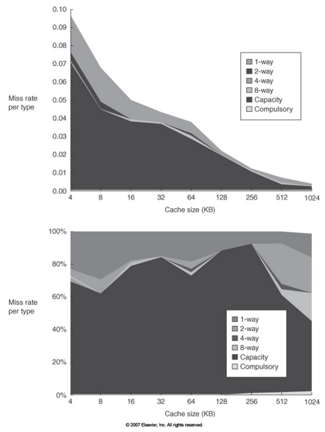
Longer miss penalty, longer hit time (address decode).
More complexity; power, expense.

Technique 3

Higher associatively

Reduces miss rate.

No point in going beyond 8 way



Heuristic: Direct mapped cache size N has the same miss rate as a two way associative cache of size N/2.

Greater hit time.

Multilevel caches reduce miss penalty

Choice is larger cache – miss rate decrease

Faster cache – miss penalty decreases.

First level cache small enough to match clock cycle.

Second level cache faster than main memory

We immediately have a problem comparing different features

Performance analysis becomes far more complicated

<memory access time> = $H_1 + M_1 * R_1$

This assumes that the miss penalty is the extra time for a cache miss – time to access for a miss is $H_1 + M_1$

$M_1 = H_2 + M_2 * R_2$

$<t> = H_1 + H_2 * (1 – R_2) + M_2 * R_2$

Multilevel caches (i)

Local Miss Rate: The number of misses divided by the total number of accesses to the cache. R_{L1} & R_{L2}

Global Miss Rate: The number of misses divided by the total number of accesses by the processor R_{L1} & $R_{L1}*R_{L2}$

Be careful about interpreting the local miss rate at level 2. If the programme fits into Level 1 cache, the only accesses to the second level cache may be the compulsory ones. R_{L2} would then be 100%.

Level 2 is best assessed via the global miss rate.

Combining reads and write times into a global average

Better is <Average memory stalls/instruction>=S

$S = R_{L1} * H_{L2} + R_{L2}*M_{L2}$

Cache requirements

L1 cache needs to be accessible in one clock cycle. Complications & size need to be restricted so that the cycle time can be as fast as possible.

L1 affects the clock time.

L2 speed only affects the *Miss penalty*. It needs to be large (L1 is a subset of L2).

Having two caches breaks the connection to the clock rate and allows more flexibility.

Useful for cache coherency

Should Level 2 cache include all the contents of Level 1 cache? *Multilevel inclusion.*

Multilevel inclusion

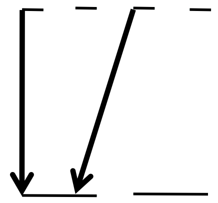
Implies

L1 much bigger than L2 ⇒ local miss for L2 will be high.

If not in L1, then not likely to be in L2. L2 only a little bigger than L1. If in L1 no reference to L2

L2 sees the compulsory misses and L1 picks up the repeat references.

Potential to invalidate several L1 blocks



Should the L1 block size be the same as the L2?

Architecturally it often looks as if L1 should be small blocks, L2 large ones. So no …

Pentium	
L1	64 bytes
L2	128 bytes

For ease yes …

If not	L1 gets a cache miss

L2 gets a cache miss

L2 fetches from memory and replaces

Several L1 blocks are now not valid

Increases L1 miss rate.

L1/L2 interactions may affect miss rates.

Multilevel exclusion

Used especially where L2 is not that much bigger than L1.

L1 data is never found in L1 and vice versa.

e.g. AMD Opteron

If L1 data is never found at L2.

So a cache miss in L1 and a hit in L2, causes a block in L1 to be exchanged with the required block at L2.

L2 will have far fewer hits than L1

(*If not miss rate at L1 must approach L2)*

L1 should optimise hit time,

While L2 needs to minimise miss rate.

L1 and L2 should not be optimised in isolation

i7 has three levels of cache.

L1 32kB: L2 256kB: L3 8Mb

Local cache and global cache. Coherency

Optimisation

Write through cache

Include a write buffer, so CPU does not have to wait for write to complete.

n) Store value in a memory location which corresponds to a block of a level 1 caches block.

Fill write buffer and proceed

n+1) Load value from a memory location which maps to the same cache block. Cache miss, fetch from level 2

n+2) Load memory location written at n), another cache miss (updated value). Miss at L1 and L2. Go to memory and load. Has the write buffer completed?

Solution: with a read miss either

wait for the write buffer to empty

or check the addresses in the write buffer.

Avoid address translation during indexing of the cache to reduce hit time

The addresses of the programme on disk are virtual addresses. They must be translated to a physical address in order to be placed in memory.

Suppose instead of doing a translate to physical address and use the physical address to determine the cache position we use the virtual address!

That still leaves open which address we use for the tag.

Both address and tags and we have a

virtual cache.

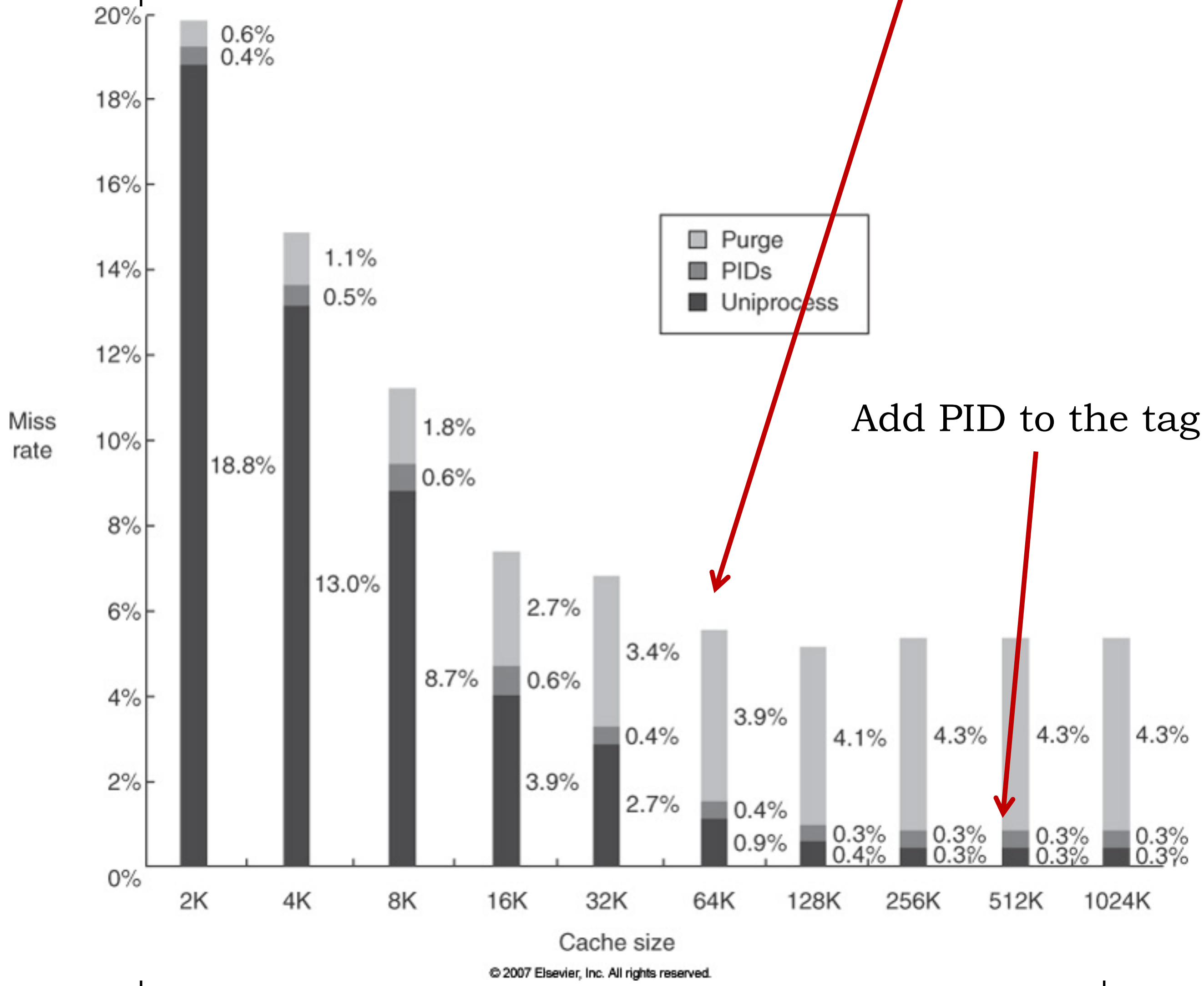
But with a virtual cache every context switch changes the virtual ⇔ physical mapping.

So the cache must be flushed.

Effect of purge on miss rate

Flushing the cache on context switching.
reduces the effect of cache size.

Add PID to the tag



Tag the cache with the Processor ID

Increase the width of the tag and include the PID of the process that wrote it.

No purge – check PID and if the same use. Otherwise signal a miss and read the correct block

Finally the OS (or the programme) may refer to the same physical location with two different virtual addresses. This can lead to inconsistencies

This can either be solved in hardware or software.

Finally I/O is usually done on physical addresses, so translation must be done there as well.

Here optimising the common case may cause significant complications and slow downs elsewhere.

Simplify the L1 cache

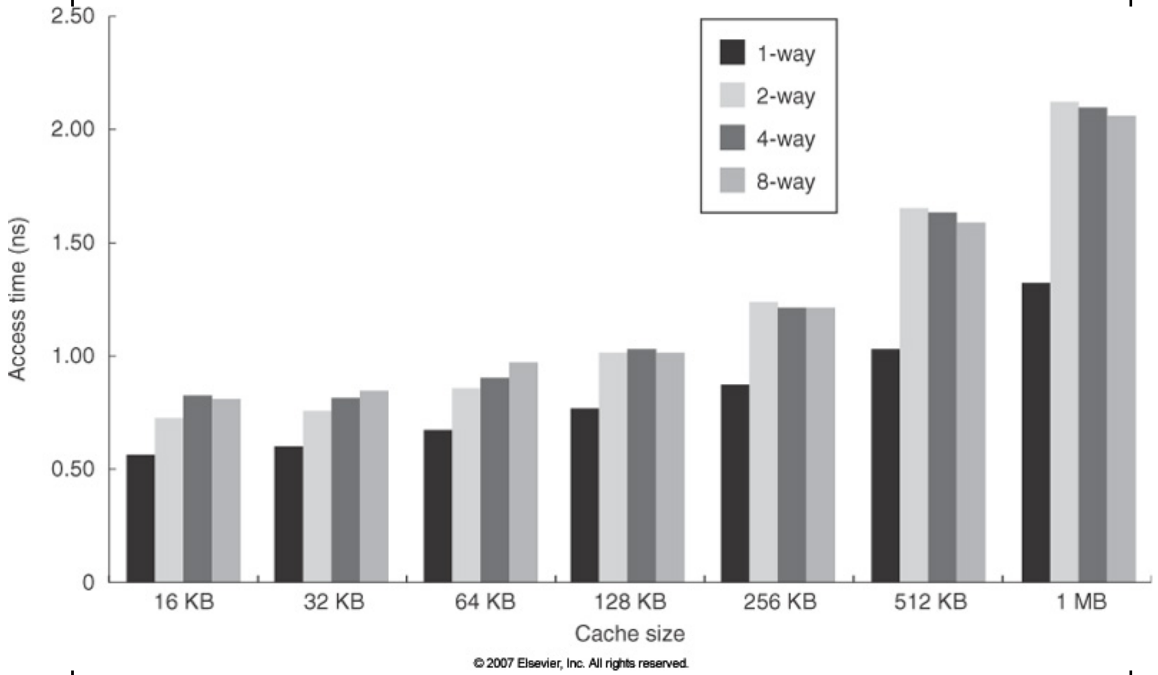
Time consuming part:

index portion of the address to read tag memory and compare it to address.

Smaller hardware faster …

Simpler hardware faster …

L1 optimise hit time



Increase cache speed rather than size and hiding **L1 misses with dynamic execution.**

Simplify the caches

Keep L2 small enough to fit on chip.

Off chip communication slower

Direct map cache is the simplest,

overlap tag check with data transmission

Example p295

Can keep tags on chip and data off chip

Compromise

Slower L1 means slower clock cycle, which slows everything, not just memory accesses./

Way prediction: reduces hit time

Each block in cache has block predictor bits.

Which block to access on the ***next*** cache access.

Multiplexor set early for block selection.

A single tag comparison performed in parallel with reading the cache data.

A miss and normal checking is done on the next cycle.

Can get accuracy considerably above 50%.

See pipelining

Used in speculative processors which undo other actions.

Pipeline cache access

Higher latency, but higher throughput.

Faster clock cycle, but greater penalty on wrong branch predictions.

Another example of optimising the common case.

Non blocking cache

If the processor does not stall on a cache miss,

It can proceed

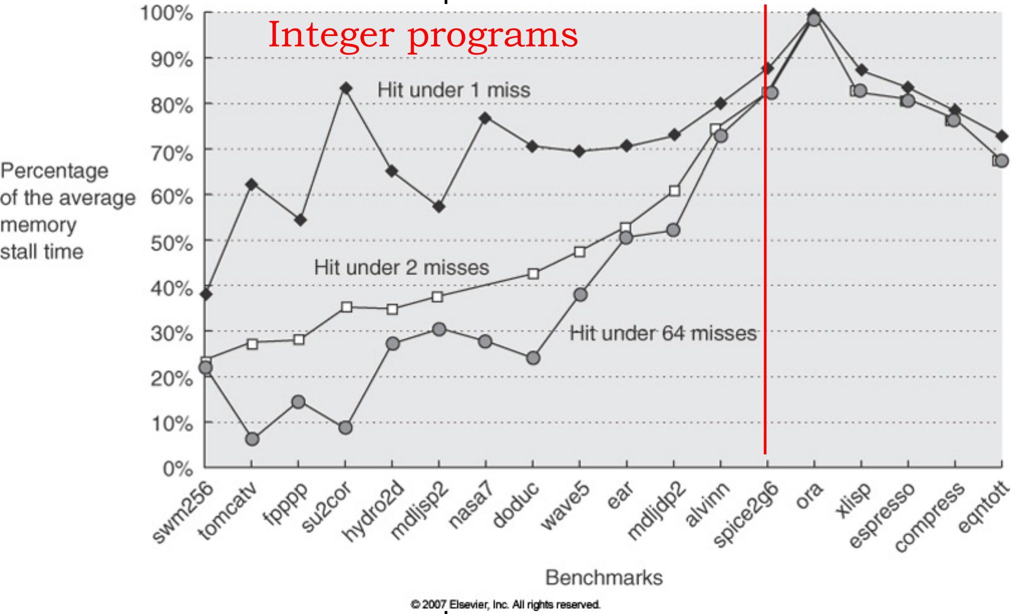
if the cache can continue to supply data while reading from L2cache/memory

Non-blocking / lockup-free cache.

"hit under miss" reduces the effect of a miss.

Overlapping multiple misses is even better

CPUs that support out of order completion.



But the memory must be able to supply more than 1 misses.

8kB data cache
spec92

Non blocking cache evaluation

Reduces L1 miss penalties, but no effect on L2

Cache miss may or may not ⇒ processor stall

Calculating the miss penalty is difficult.

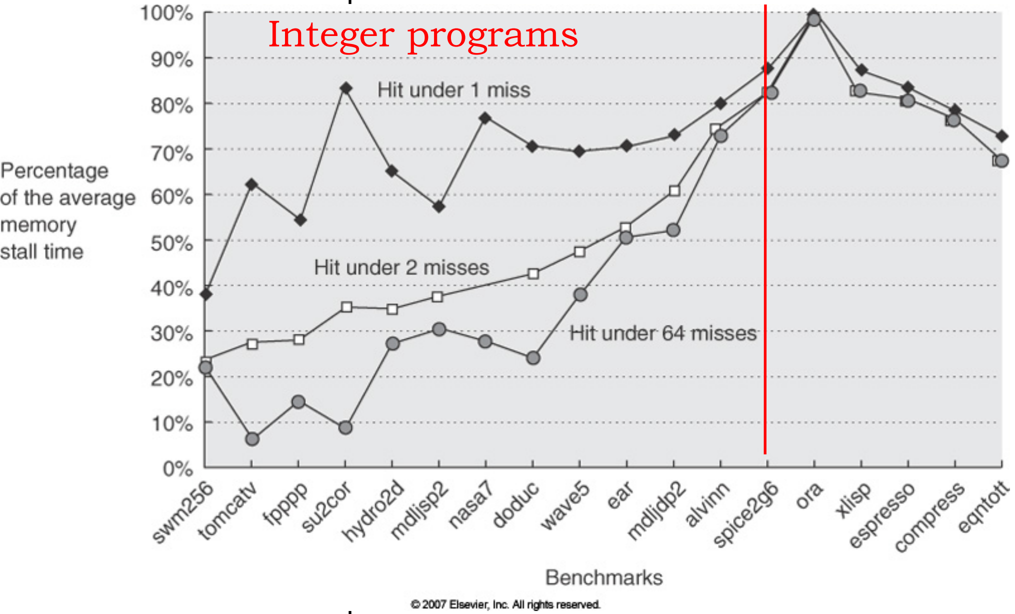
So we cannot (reliably) calculate the average memory access time.

Effect depends on:

a) miss penalty for multiple misses

b) Memory reference pattern ⟷ The more complex the CPU, the more the application becomes the benchmark

c) Processor performance with outstanding misses.



Non the less the gains are real, as are the complexities.

8kB data cache
spec92

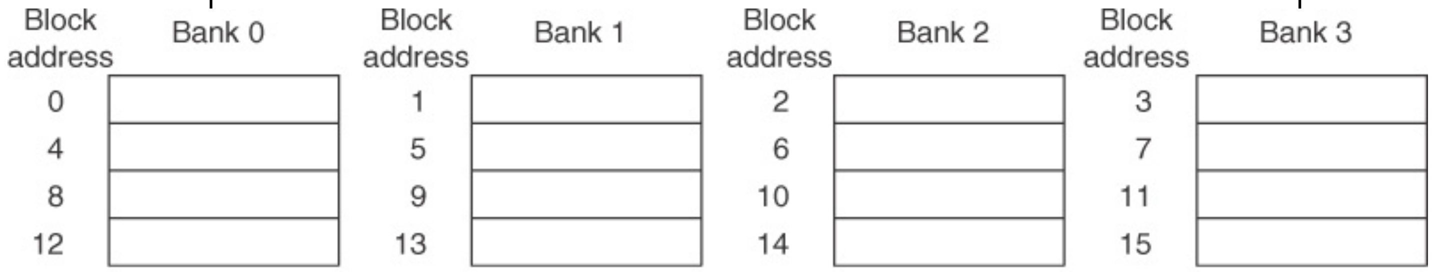
Multi-banked caches

Divide cache into multiple *banks.* Each one can be individually addressed.

Organise banks so multiple paths can be used.

One way is *sequential interleaving.*

If you want to access a block = 1 modulo 4, you want to be able to do it with a single access.



Start from a miss as soon as possible

A miss transfers a block or words

(penalty is latency + blocksize/bandwidth)

Likely to be useful with large blocks

(normally) the processor needs just one word to restart.

So either transfer the word to register as soon as it arrives *early restart* and continue filling block while CPU resumes.

Or explicitly get the requested word first *critical word first* and again continue to fill.

More complex … better?

Locality ⇒ other words in the block are likely to be needed as well – actually if the access are sequential in the block there may be little difference between the two methods

Technique13

Merge write buffers

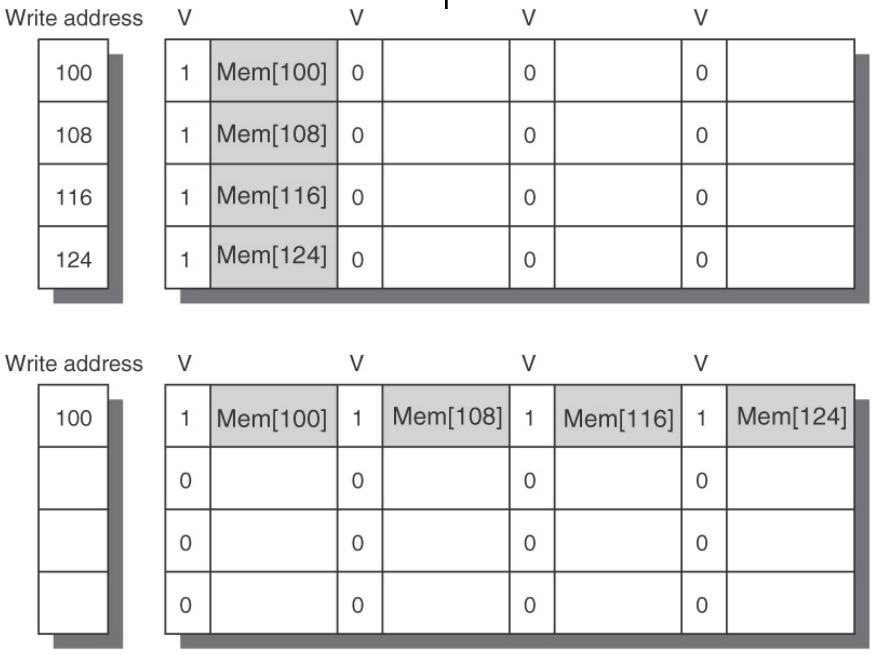
Write through cache and write back cache both use *write buffers*.

For a write buffer with space the data and physical address are written. CPU is finished and the buffer starts the writing process.

If the buffer is full the CPU has to wait.

On write, check if an address is already in a block waiting to be written (*temporal locality*). If there is merge. More updates done on a single write, less chance of the buffer filling.

Compromise



Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Merging problematical with I/O buffers, because the I/O registers do not act like a block of memory.

Includes reducing miss penalties (no stall on buffer full). Does not fit into the taxonomy.
Not optimising the common case. Writes are rare. Although merging depends on common case. Writes are sequential.

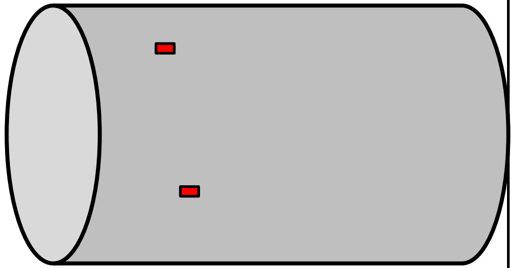
Optimisation

Compiler optimisation

In the early days programmers optimised memory accesses by hand.

Drum memory long latency

arrange words for sequential access so that the next word is under the read head when required.



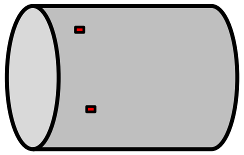
The programmers built the machine.

Now the manufacturers will provide compilers to do analogous things.

Warning

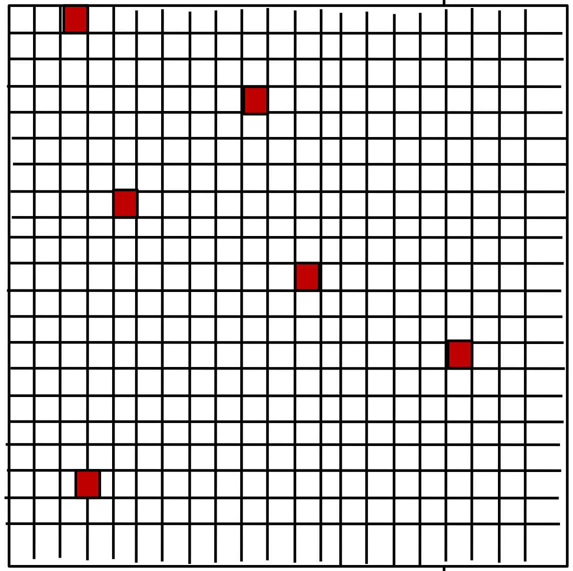
Benchmarks are well established, stable, a sitting target. Manufacturers write compilers which return good performance for benchmark programs. (may even have special switches).

Compiler optimisation

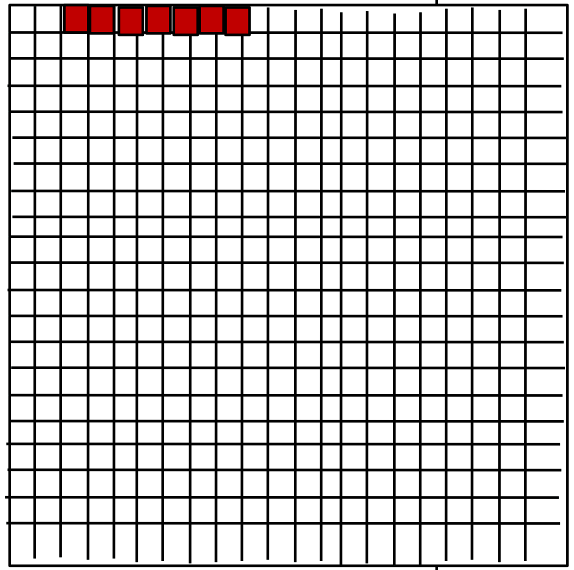


Loop interchange

```
for (int j=0; j < 100;  j++) {
    for (int i=0; i <5000; i++) {
        x[i][j] = 2 * x[i][j]
    }
}
```



```
for (int i=0; i < 100;  i++) {
    for (int j=0; j <5000; j++) {
        x[i][j] = 2 * x[i][j]
    }
}
```



Compiler optimisation

Block alignment

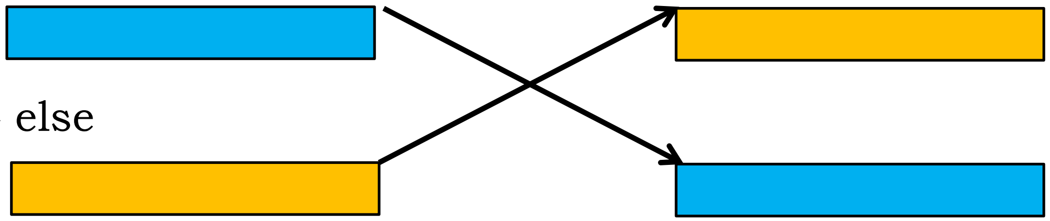
```
for (pnt=0; pnt < 400;  pnt++) {
........
......
}
```

If the compiler puts the beginning of the loop at the beginning of a cache block we are likely to load many of the instructions we need with a single load

Branch straightening

if (   ) then {

Code Block



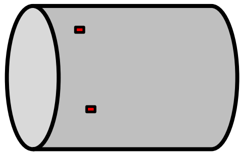
} else

Code Block

}

If the else block is normally taken, then reorder so code immediately follows if in memory.

Likely to be in same cache block



Compiler optimisation

Blocking

```
for (int row=0; row < N;  row++) {
  for (int col=0; col < N; col++) {
    for  (int pnt=0; pnt<N; pnt++)
       x[i][j] = x[i][j] + y[i][k]*z[k]j]
 }}}
```

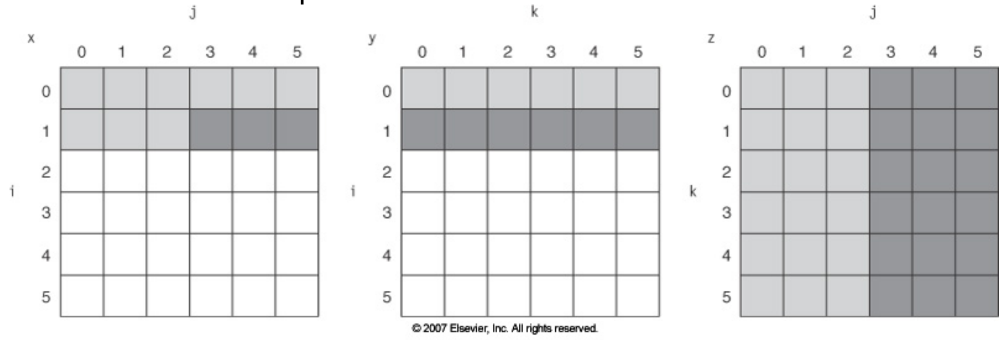
Neither row order nor column order will work here.

All three matrices should ideally be held in cache.

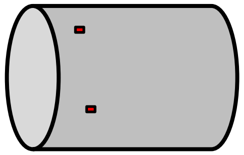
Want the **row** of y and the **col** of z as a minimum.

If the cache can hold submatrices of size B by B

Step through in those sizes.



Light: old accesses
Dark: recent accesses
White: not touched

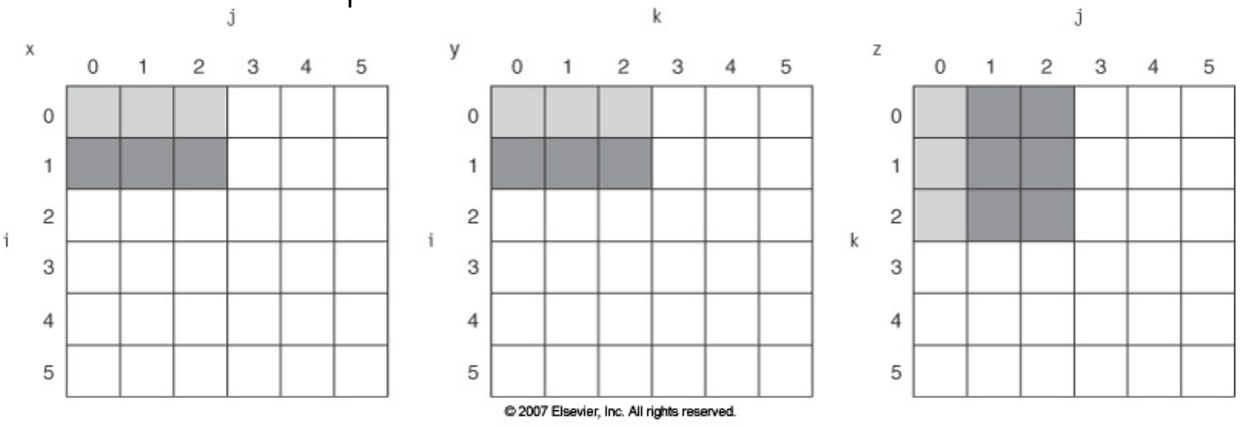


Compiler optimisation

```
for (int bCol=0; bCol<N;bCol = bCol = bCol+B) {
  for (int bPnt=0; bPnt<N;bPnt=bPnt+B) {
    for (int row=0; row < N;  row++) {
      for (int col=bCol; col < min(bCol+B,N); col++) {
       for  (int pnt=bPnt; pnt<min(bPnt+B,N); pnt++)
          x[i][j] = x[i][j] + y[i][k]*z[k]j]
 }}}}}
```

y benefits from spatial locality and z from temporal.

These techniques go back to pre-cache days where people would *hand code* inner loops to improve performance in critical areas. Keeping variables in the registers



Light: old accesses
Dark: recent accesses
White: not touched

Compiler controlled prefetching

Miss penalty or Miss rate.

Compiler analyses code. Identifies instructions and data which can be loaded before needed.

Inserts instructions into the code which causes these to be loaded ahead of time.

Register prefetch

Cache prefetch

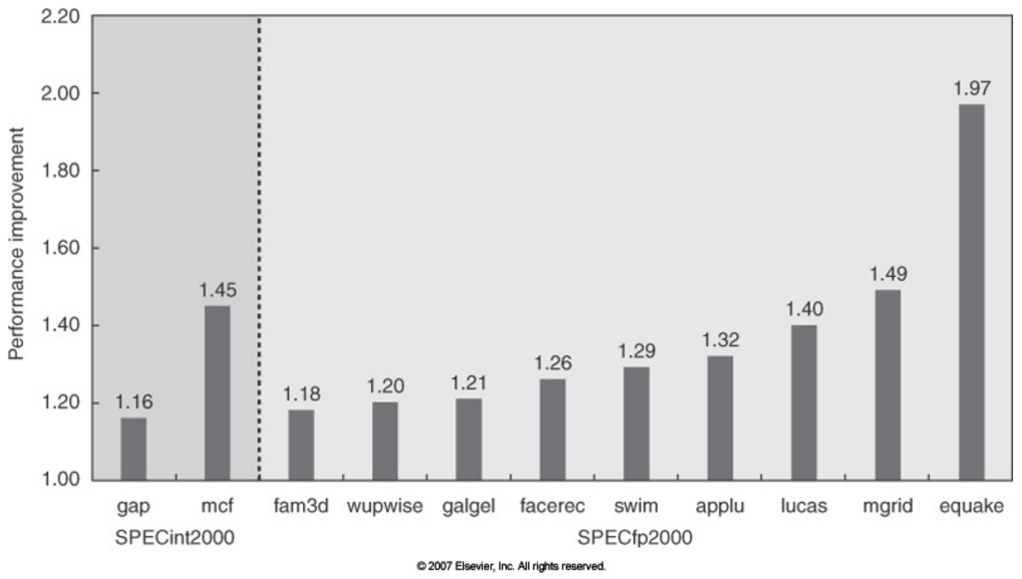
Either can be *faulting* or *non-faulting (non-binding)*

Non-faulting turn into nops if they would cause an exception.

We want prefetches to operate silently. They must not interfere with the normal running of the processor. They should not change contents of registers and memory and cannot cause virtual memory faults.

Normal for modern machines

Compiler controlled prefetching



Prefetch speedup for Pentium 4. The best results for specint2000 (2/12) and specfp2000 (9/14).

Overlap execution and fetch.

Loops are a good target.

Either:

loop unroll a couple of times

Miss penalty small

Or

loop unroll many times or software pipelining

Miss penalty large

Out of order CPUs

Execution may be possible during cache miss.

Pending store stays in the register (or special register.

Instructions which require results of instructions affected by cache miss, wait to start execution.

Independent instructions continue.

Misses have an effect very difficult to estimate or even measure.

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{total miss latency} - \text{overlapped miss latency})$$

Depends on program, depends on compiler efficiency.

Depends on algorithm

Quick sort v Radix Sort

Algorithmically radix sort is better.

Visible for large number of items

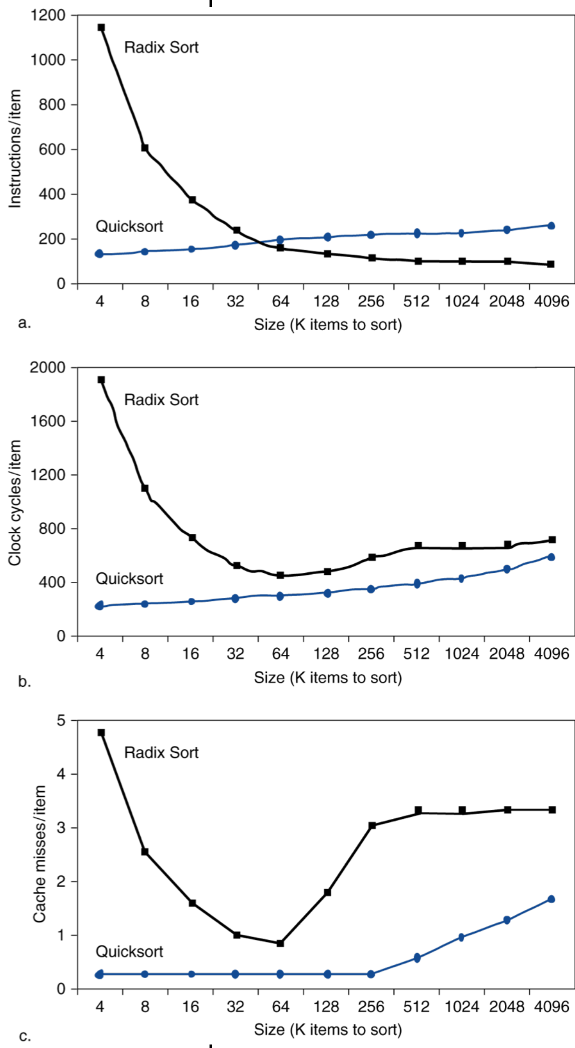
Advantage disappears for low number of items.

Looking at the numbers for instructions shows cross over at low items.

Looking at Clock cycles – there is never an advantage for Radix sort.

Looking at Cache misses shows where the problem is coming from.

New versions of radix sort which allows for cacheing and works faster



Sort based on comparing digits in the same significant position of the number. Traced back to Hollerith's work on US census.

Summary

Technique	Hit time	Miss Rate	Miss penalty
Increase block size		X	
Increase cache size		X	
Higher associativity		X	
compiler optimisation		X	
Parallelism		X	X
prefetching (hardware and compiler)		X	X
Multilevel caches			X
Write through cache			X
Critical word first			X
Merge write buffers			X
Increase cache bandwidth			X
Simple caches	X		
Way Prediction	X		
Avoid address translation	X		
Non blocking cache			X

Piled higher and Deeper

Companies that make golf equipment make:

Shafts which hit the ball further;

Heads which hit the ball further;

Balls that fly further and straighter.

Better tees … really!

http://www.golfonline.co.uk/links-choice-10-booster-golf-tees-p-3207.html

Technical improvements

If we believe that their affects are multiplicative then we ask why Rory McIlroy only hits the ball 300m.

Actually (even if we believe they work) we realise that they will not all work together.

Cache optimisations cannot be piled on top of each other.

Some of them work well together – others less so.

Others are even antagonistic.

Optimisation is hard

AMD and Intel do not follow the same strategies.

There have been some notable errors from each.