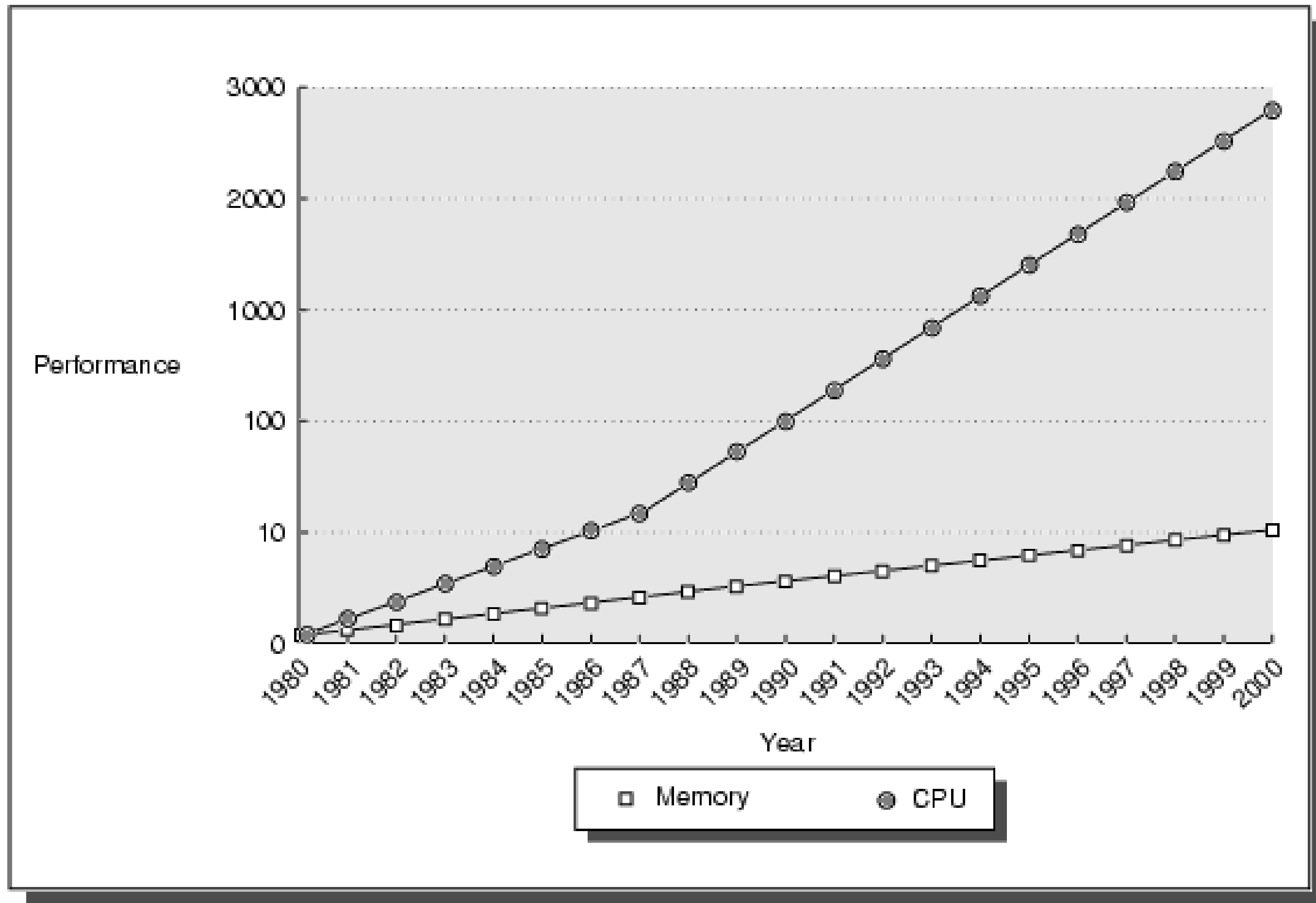
Why memory-hierarchy ?



Why memory-hierarchy ?

Principle of locality

- Temporal locality
- Spatial locality

The slower the memory the cheaper(Basic hw rule!)

Memory hierarchy measures

Memory stall cycles = IC x MRP x MR x MP

Where :

IC = Instruction Count

MRP = Memory References per Instruction

MR = Miss Rate

MP = Miss Penalty

Cache = Buffers/small memories that are applied to reuse commonly occurring items

4 Basic questions about caches

- **Block placement**

  Where can a block be placed in the upper level?
- **Block identification**

  How is a block found if it is in the upper level?
- **Block replacement**

  Which block should be replaced on a miss?
- **Write Strategy**

  What happens on a write?

Where can a block be placed?

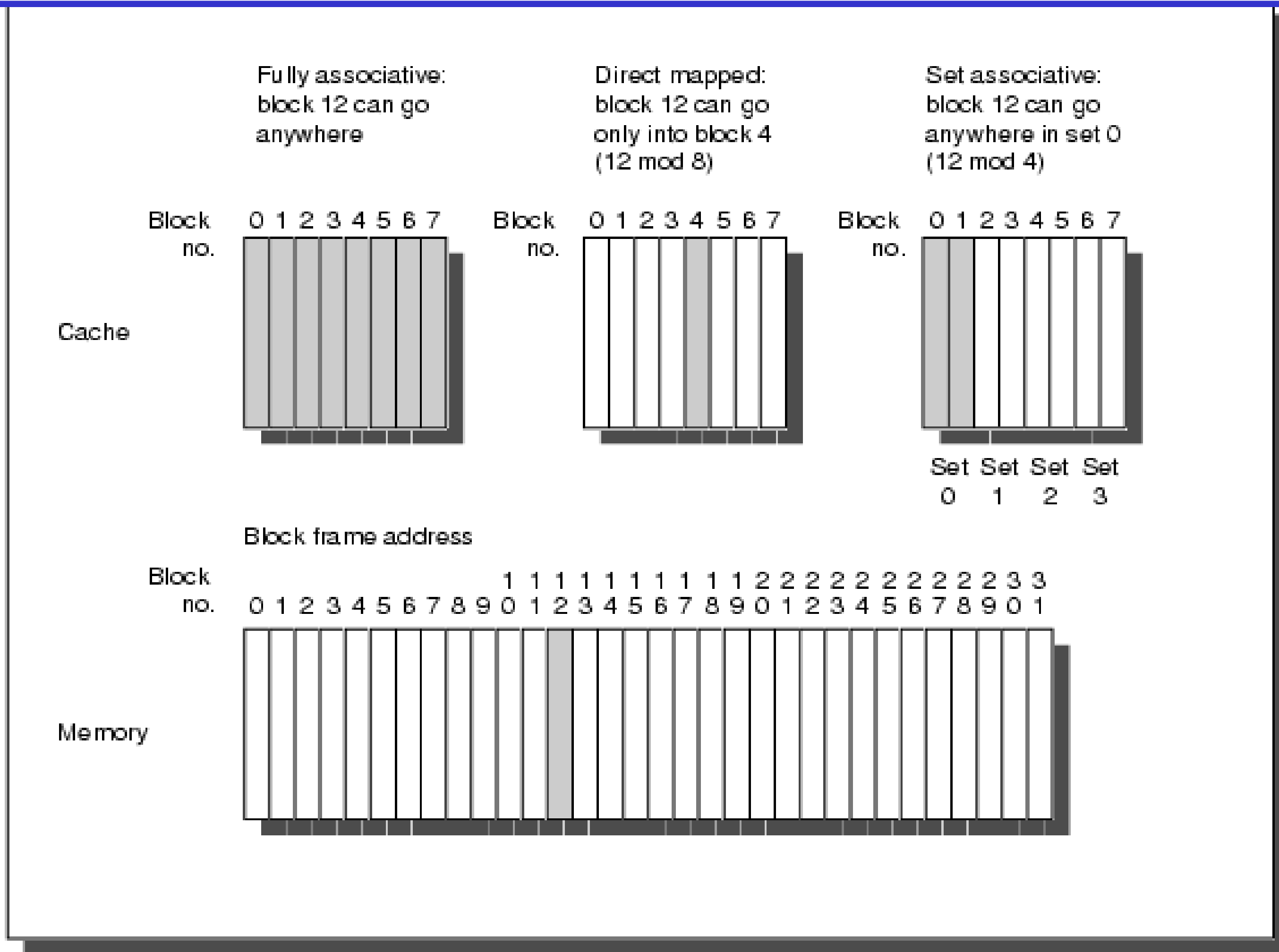
- **Direct Mapped**

  In only one specific place.
- **Fully Associative**

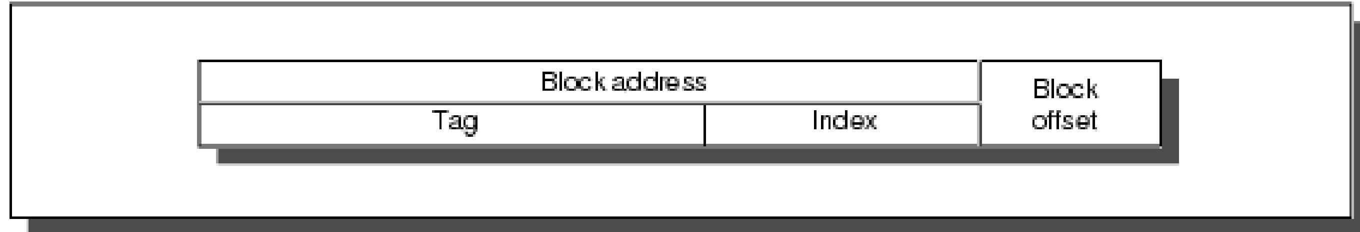
  Anywhere !
- **Set Associative**

  In a restricted set of places (in a set).

Where can a block be placed?



How is a block found?



Tag : Which block in a set.

Index : Which set.

Block Offset : Address of data within the block.

When Associativity increases -> Tag increases and Index decreases

$$2^{index} = (\text{Cache size}) / (\text{Block size} * \text{Set associativity})$$

Which block should be replaced?

Random: In practice it is pseudorandom

LRU: Least recently used (i.e. unused for a long time)

	Associativity					
	Two-way		Four-way		Eight-way	
Size	**LRU**	**Random**	**LRU**	**Random**	**LRU**	**Random**
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

What happens on a Write ?

Reads much more common than writes so :

Amdahl's law:

"Make common case fast" -> Make reads fast

So: Writes are (much) slower and much more complicated

- **Write through**

    Info is written to both cache and memory

- **Write back**

    Info is written only to the cache. Modified block is written to the memory only when replaced

What happens on a Write ?

- **Write through**
  - +Easier.
  - +Never delay a read.
  - +Memory has always an up-to-date copy
  - - More memory bandwidth
- **Write back**
  - +Less Bandwidth
  - -Complicated

What about write misses ?

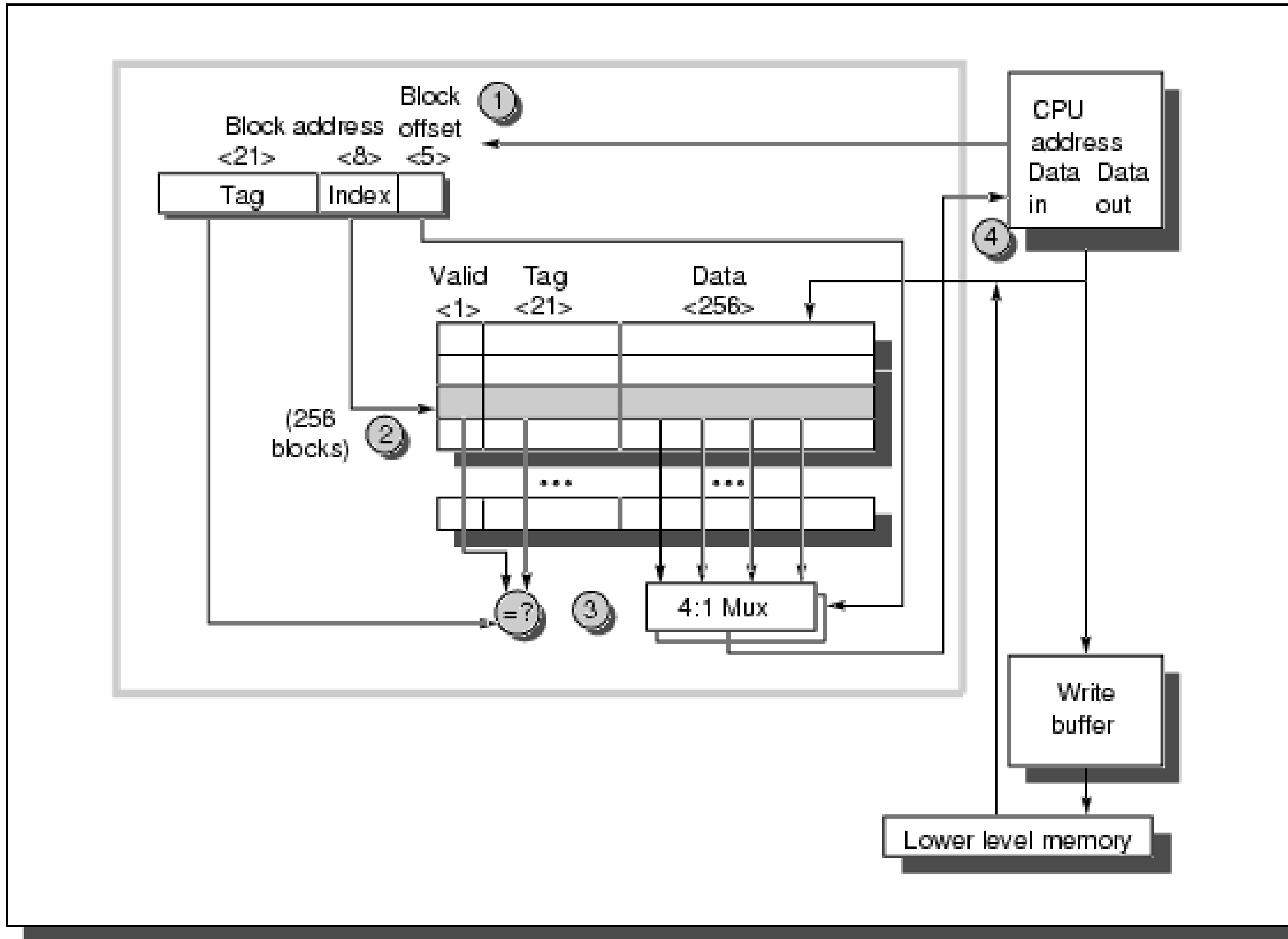
- **Write allocate**

  Block is loaded to the cache and then written

- **No-write allocate**

  Block is not loaded, it is modified only in the memory

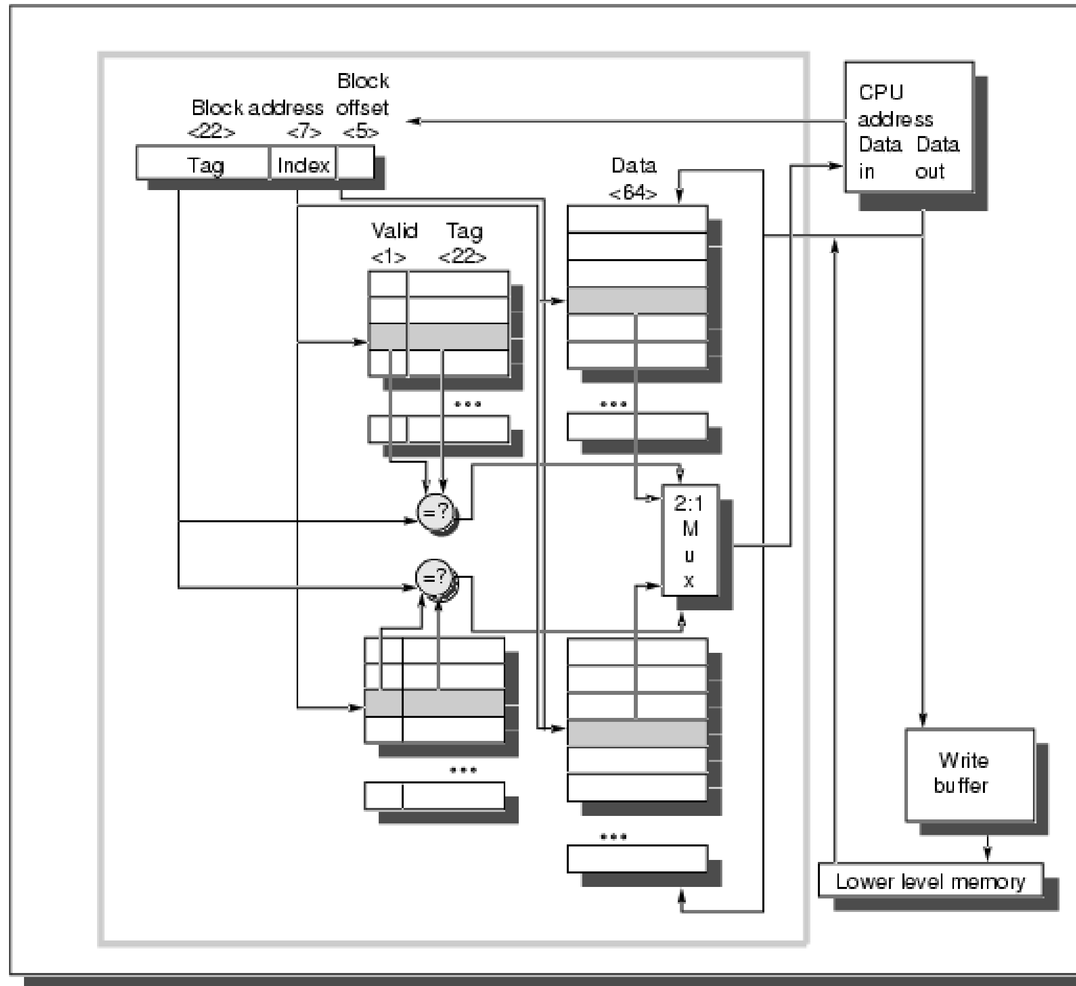
An example:Alpha AXP 21064



8KB Cache

32-byte Block

21064 2-way Set Associative



Cache performance

MemAccTime = HitTime + Miss rate x Miss Penalty

Size	Instruction cache	Data cache	Unified cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%

Example 1

Which has the lower miss rate: A 16-KB instruction cache with a 16-KB data cache or a 32-KB unified cache ? Assume a hit takes 1 clock cycle on the separate caches and miss costs 50 clock-cycles. On the unified cache the hit takes 2 clock cycles since the memory is single ported. What is the Average Memory Access time ? (Ignore write stalls)

Cache Performance (2)

CPU time = (CPC + MSC) x CP

CPC = CPU execution clock cycles

MSC = Memory stall clock cycles

CP = Clock period

MR= Miss Rate MP = Miss Penalty

MSC = MemAcc x MR x MP =>

CPI time = IC x CP

(CPI + (MemAcc/Instr) x MR x MP)

Example 2

Impact of different cache organizations

Assume 2 caches a Direct Mapped and a 2-way Set Associative :

Assume a CPI with a perfect Cache is 2.0, a CP of 2ns, there are 1.3 MemRefs per instuction and the size of both caches is 64KB (32 bytes blocks). In the 2-way S.A. CP is 1.1 times larger due to the additional hardware. Miss penalties are the same in both cases (70ns). The Miss Rate of the Direct-Mapped is 1.4% and of the 2-way S.A. is 1%

Improving Cache Performance

MemAccTime = HitTime + Miss rate x Miss Penalty

- **Reducing the miss rate**
- **Reducing the miss penalty**
- **Reducing the hit time**

Cache Misses

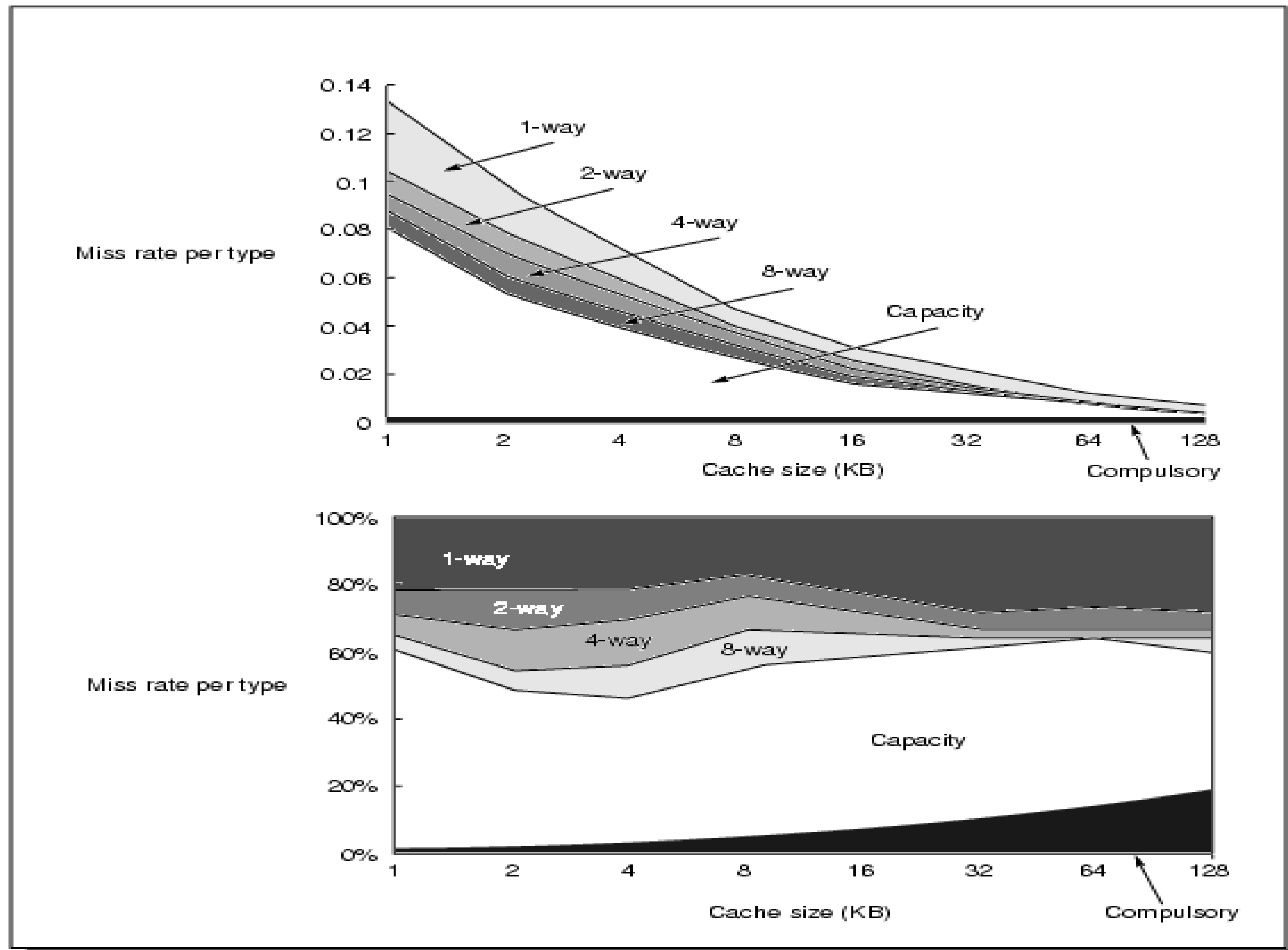
Types of misses :

- **Compulsory**
  First access to a block
- **Capacity**
  Not enough space
- **Conflict (**only on set associative or direct mapped**)**
  Too many blocks map to the same set

Cache Misses

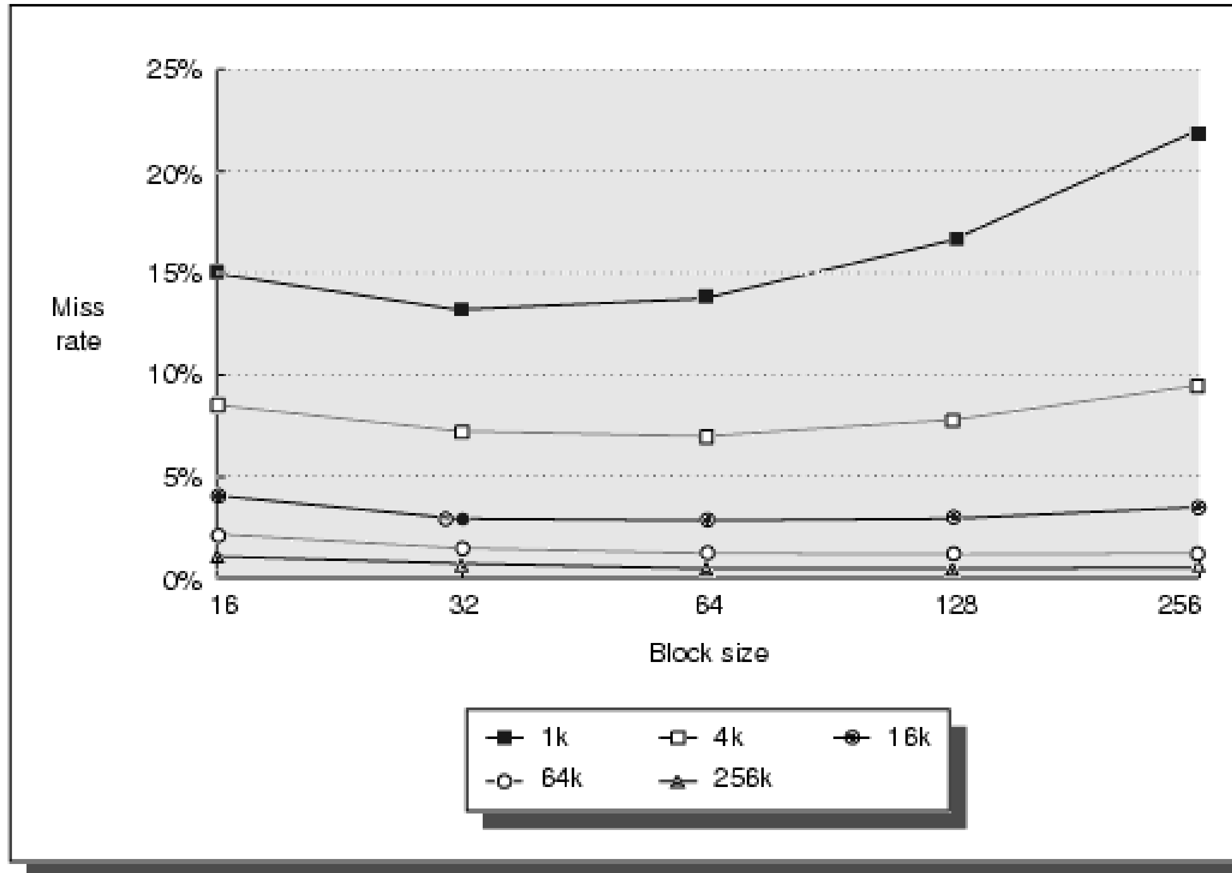
Cache size	Degree associative	Total miss rate	Miss rate components (relative percent) (*Sum = 100% of total miss rate*)					
			Compulsory		Capacity		Conflict	
1 KB	1-way	0.133	0.002	1%	0.080	60%	0.052	39%
1 KB	2-way	0.105	0.002	2%	0.080	76%	0.023	22%
1 KB	4-way	0.095	0.002	2%	0.080	84%	0.013	14%
1 KB	8-way	0.087	0.002	2%	0.080	92%	0.005	6%
2 KB	1-way	0.098	0.002	2%	0.044	45%	0.052	53%
2 KB	2-way	0.076	0.002	2%	0.044	58%	0.030	39%
2 KB	4-way	0.064	0.002	3%	0.044	69%	0.018	28%
2 KB	8-way	0.054	0.002	4%	0.044	82%	0.008	14%
4 KB	1-way	0.072	0.002	3%	0.031	43%	0.039	54%
4 KB	2-way	0.057	0.002	3%	0.031	55%	0.024	42%
4 KB	4-way	0.049	0.002	4%	0.031	64%	0.016	32%
4 KB	8-way	0.039	0.002	5%	0.031	80%	0.006	15%
8 KB	1-way	0.046	0.002	4%	0.023	51%	0.021	45%
8 KB	2-way	0.038	0.002	5%	0.023	61%	0.013	34%
8 KB	4-way	0.035	0.002	5%	0.023	66%	0.010	28%
8 KB	8-way	0.029	0.002	6%	0.023	79%	0.004	15%
16 KB	1-way	0.029	0.002	7%	0.015	52%	0.012	42%
16 KB	2-way	0.022	0.002	9%	0.015	68%	0.005	23%
16 KB	4-way	0.020	0.002	10%	0.015	74%	0.003	17%
16 KB	8-way	0.018	0.002	10%	0.015	80%	0.002	9%
32 KB	1-way	0.020	0.002	10%	0.010	52%	0.008	38%
32 KB	2-way	0.014	0.002	14%	0.010	74%	0.002	12%
32 KB	4-way	0.013	0.002	15%	0.010	79%	0.001	6%
32 KB	8-way	0.013	0.002	15%	0.010	81%	0.001	4%
64 KB	1-way	0.014	0.002	14%	0.007	50%	0.005	36%
64 KB	2-way	0.010	0.002	20%	0.007	70%	0.001	10%
64 KB	4-way	0.009	0.002	21%	0.007	75%	0.000	3%
64 KB	8-way	0.009	0.002	22%	0.007	78%	0.000	0%
128 KB	1-way	0.010	0.002	20%	0.004	40%	0.004	40%
128 KB	2-way	0.007	0.002	29%	0.004	58%	0.001	14%
128 KB	4-way	0.006	0.002	31%	0.004	61%	0.001	8%
128 KB	8-way	0.006	0.002	31%	0.004	62%	0.000	7%

Cache Misses



Make Blocks Larger

Good due to spatial locality; Bad due to less blocks



Make Blocks Larger(2)

Block size	Cache size				
	1K	**4K**	**16K**	**64K**	**256K**
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

But increases Miss Penalty !!!!

Block size	Miss penalty	Cache size				
		1K	**4K**	**16K**	**64K**	**256K**
16	42	7.321	4.599	2.655	1.857	1.458
32	44	**6.870**	**4.186**	**2.263**	1.594	1.308
64	48	7.605	4.360	2.267	**1.509**	**1.245**
128	56	10.318	5.357	2.551	1.571	1.274
256	72	16.847	7.847	3.369	1.828	1.353

Increase Associativity

- An 8-Way set associative cache is as effective as a fully-associative one.

- *2:1 cache rule of thumb* : A direct-mapped cache of size N has about the same rate as a 2-way Set Associative of size N/2.

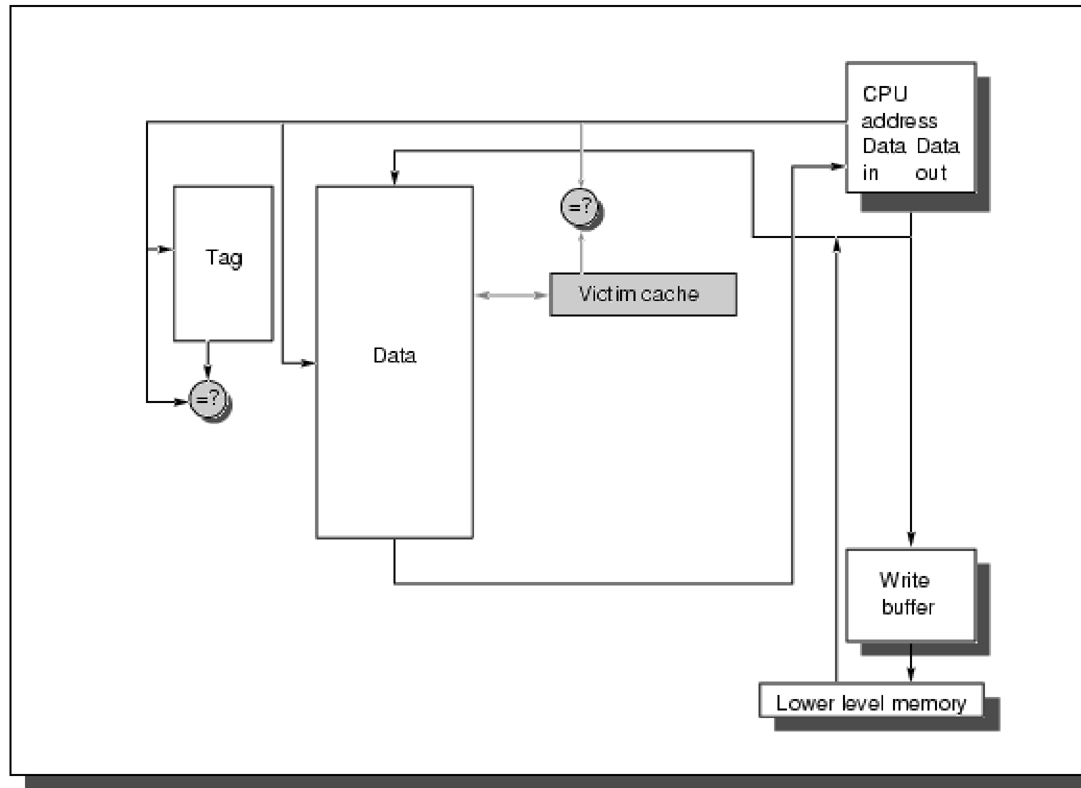
- BUT: Increases the hit time… By 2% when going from Direct Mapped to 2-way Set Associative.

Increase Associativity(2)

SO :

Cache size (KB)	Associativity			
	One-way	Two-way	Four-way	Eight-way
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	**1.79**
64	1.70	1.60	1.57	**1.59**
128	1.50	1.45	1.42	**1.44**

Victim Caches



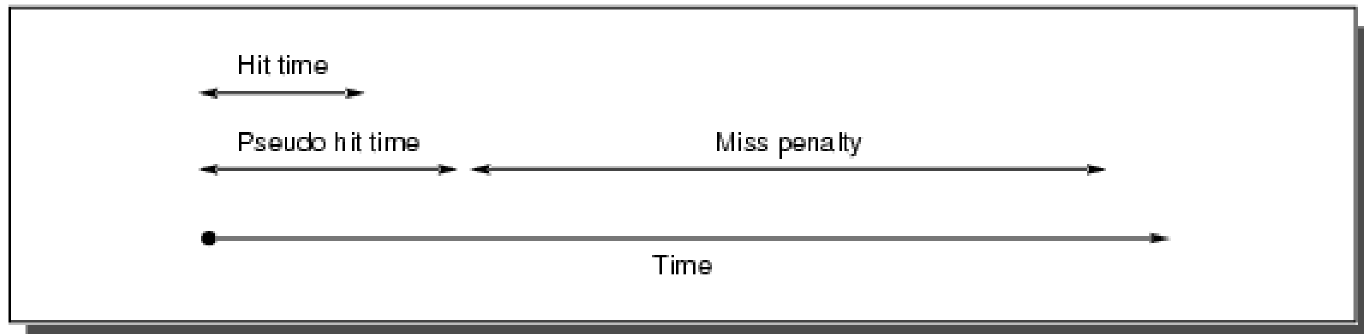
A 4 entry one can remove 20%-95% of the conflict misses in a 4-KB direct-Mapped cache!!!!

Pseudo-Associative Caches

After a read miss and before going to the next level, another cache entry is checked (I.e. invert the most significant bit of the index).

As a result we have two hit times : A fast (regular) hit time and a slow (pseudo) hit time.

Danger: Many of the fast hit times become slow.



Hardware Prefetching

On a read Instruction miss the next consecutive block is prefetched and put in the instruction stream buffer(ISB). The ISB is searched on read a miss and if the item is there then a new prefetch is issued.

- A 1-block ISB reduces misses 15%-25%
- A 4-block ISB reduces misses 50% (!!)
- A 16-block ISB reduces misses 72%

The same with Data Caches can reduce misses by 25%-70% depending on sizes of the stream buffer.

Compiler-Control Prefetching

- **Register Prefetch**: Loads value into a register.

- **Cache Prefetch**: Loads data only into the cache.

It makes sense only if the cache is *non-blocking* :
It can supply data while prefetching other things.

They are used mainly in loops and depending on
The miss penalty the compiler unrolls the loop either
a couple of times or a larger number of times.

Compiler Optimizations: Merging Arrays

Improve spatial locality :

```
/* Before */
int val[SIZE];
int key[SIZE];
--------------------------------
/* After */
struct merge {
      int val;
      int key;
};
struct merge mergedarray[SIZE];_
```

Compiler Optimizations: Loop Interchange

Improve spatial locality :

```
/* Before */
for (j = 0; j < 100; j++)
     for (k = 0; k < 5000; k++)
          x[k][j] = 2 * x[k][j];
-------------------------------------------
/* After */
for (k = 0; k < 5000; k++)
     for (j = 0; j < 100; j++)
          x[k][j] = 2 * x[k][j];
```

Compiler Optimizations: Loop Fusion

Improve temporal locality :

```
/* Before */
for (k = 0; k < N; k++)
     for (j = 0; j < N; j++)
          a[k][j] = b[k][j] * c[k][j];

 for (k = 0; k < N; k++)
     for (j = 0; j < N; j++)
          d[k][j] = a[k][j] + c[k][j];
```

Compiler Optimizations: Loop Fusion(2)

```
/* After */
for (k = 0; k < N; k++)
      for (j = 0; j < N; j++)
      {
            a[k][j] = b[k][j] * c[k][j];
            d[k][j] = a[k][j] + c[k][j];
      }
```

The original code takes all the misses to access *a* and *c* twice (once in every loop).

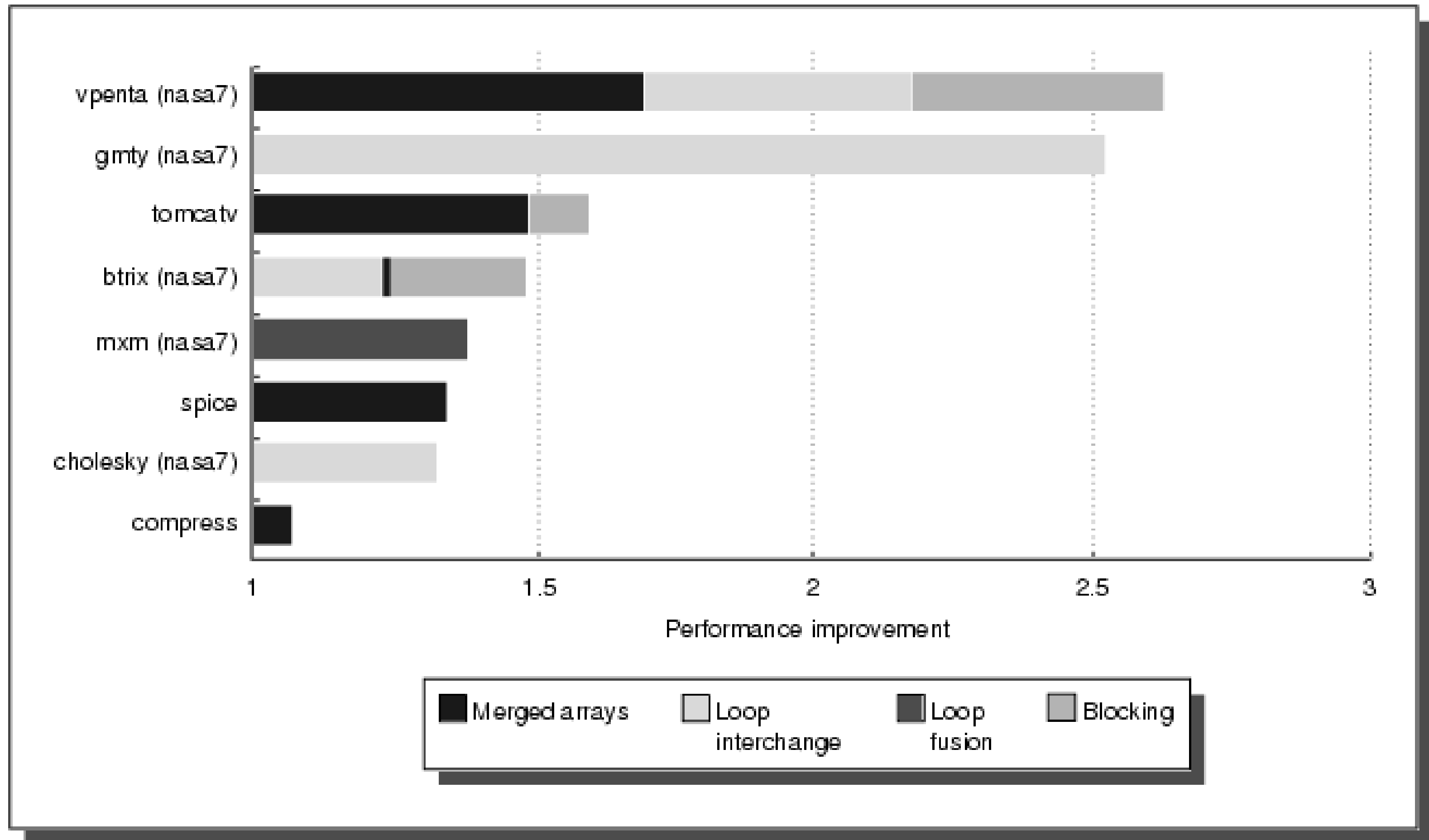
Compiler Optimizations:Blocking(Before)

```
/* Before */
for (k = 0; k < N; k++)
      for (j = 0; j < N; j++)
      {
            r = 0;
            for (l = 0; l < N; l++) {
                  r = r + y[k][l] * z[l][j];
            x[k][j] = r;
      }
```

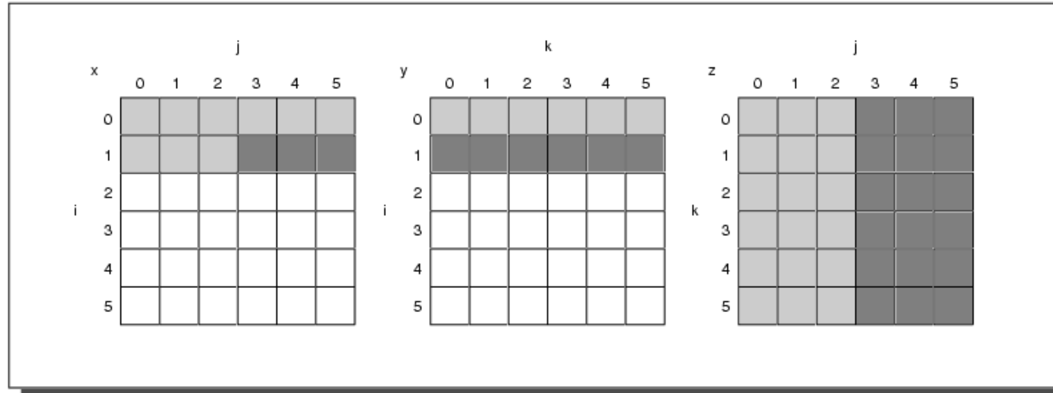
Compiler Optimizations:Blocking(After)

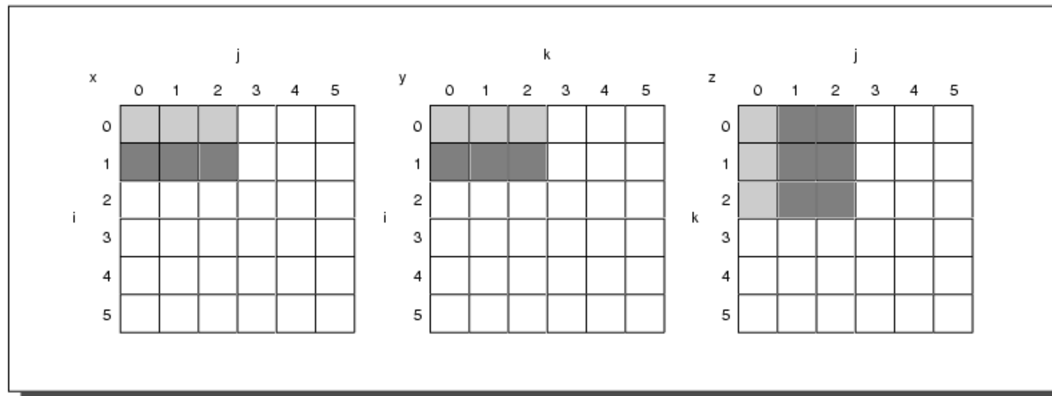
```
* After */
or (jj = 0; jj < N; jj+=B)
    for (kk = 0; kk < N; kk+=B)
        for (i = 0; i < N; i++)
            for (j = jj; j < min(jj+B-1,N);j++
            {
                r = 0;
                for (k = kk;
                    k < min(kk+B-1,N);k++)
                        r=r+y[i][k]*z[k][j];
            x[i][j] = x[i][j] + r;
            }
```

Compiler Optimizations: General Results



Compiler Optimizations: Blocking (Comparison)





Reducing Miss Penalty: Priorities

Give priority to read misses over Writes

Problem: (when there is a write buffer):

SW 512(R0),R3

LW R1,1024(R0)

LW R2,512(R0)

If not careful R2 and R3 will NOT be equal!!

So either wait until buffer is empty (1.5 times larger Miss penalty on real programs!)

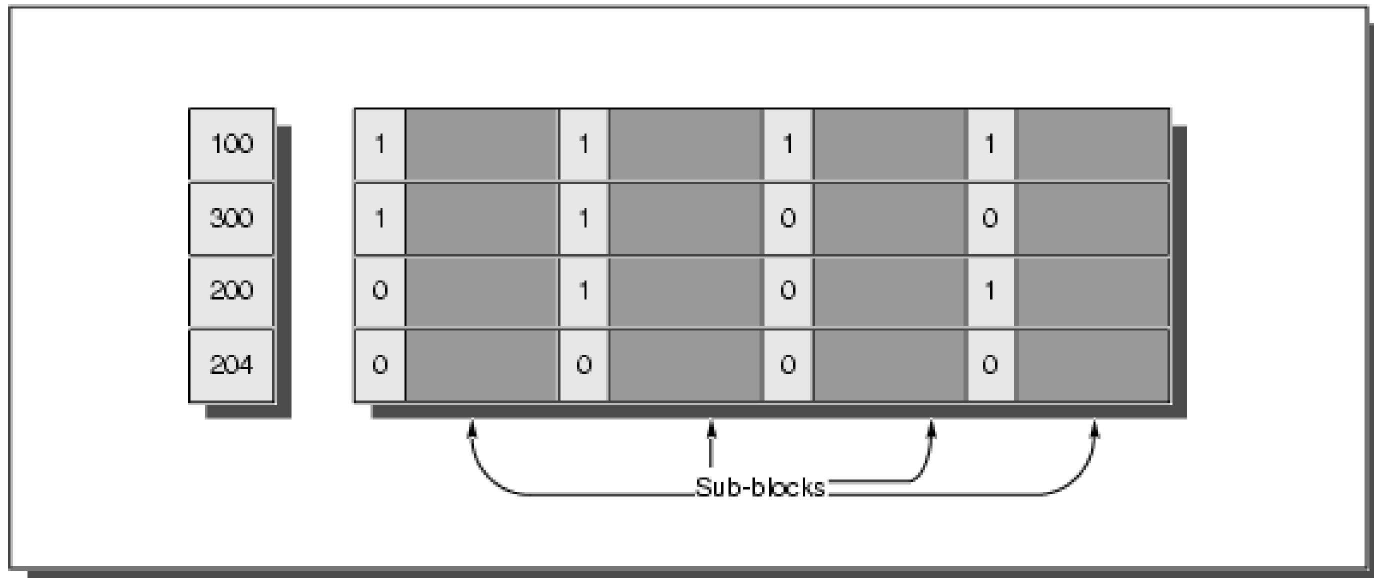
Or check contents of buffer!

Reducing Miss Penalty: Sub-blocks

Sub-block placement:

A valid bit is added to sub-blocks and only one sub-block should be read on a read miss.

Advantage : Need less tags !



Reducing Miss Penalty: Early starts

- **Early restart:**

  As soon as the requested word arrives send it to the CPU

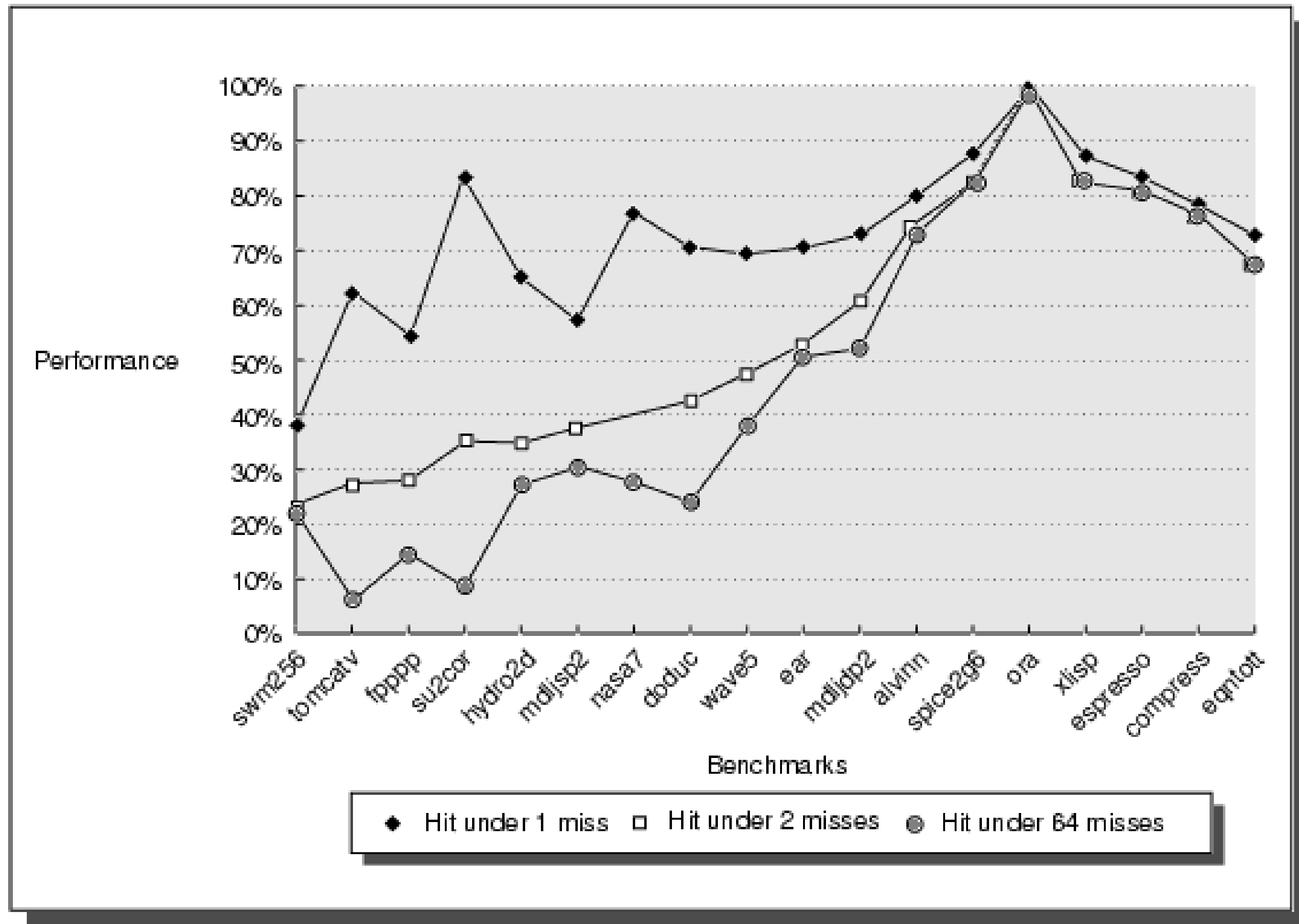
- **Critical Word First(wrapped fetch):**

  Request the missed word first and send it to the CPU; then fill the block while CPU is working.

Reducing Miss Penalty: Non-blocking caches

When used with sophisticated machines the caches should not stall on a miss. That's called "hit under miss" (e.g. A Pipelined machine with out-of-order execution using scoreboard or Tomasulo algorithm)
These are the *non-blocking or lockup-free* caches

Reducing Miss Penalty: Non-blocking caches



Reducing Miss Penalty: Second level caches

The only method which concentrates on the MEM to cache interface.

Q: Should I make the cache faster to keep pace with the speed of CPUs or make the cache larger to overcome the widening gap between the CPU and the MEM ?

A: Both ! Add a second level cache ! This will make the 1st level one small and fast and the 2nd level one large.

Second level caches

$AMAT = Hit_L1 + MR__L1 \times MP_L1$ And

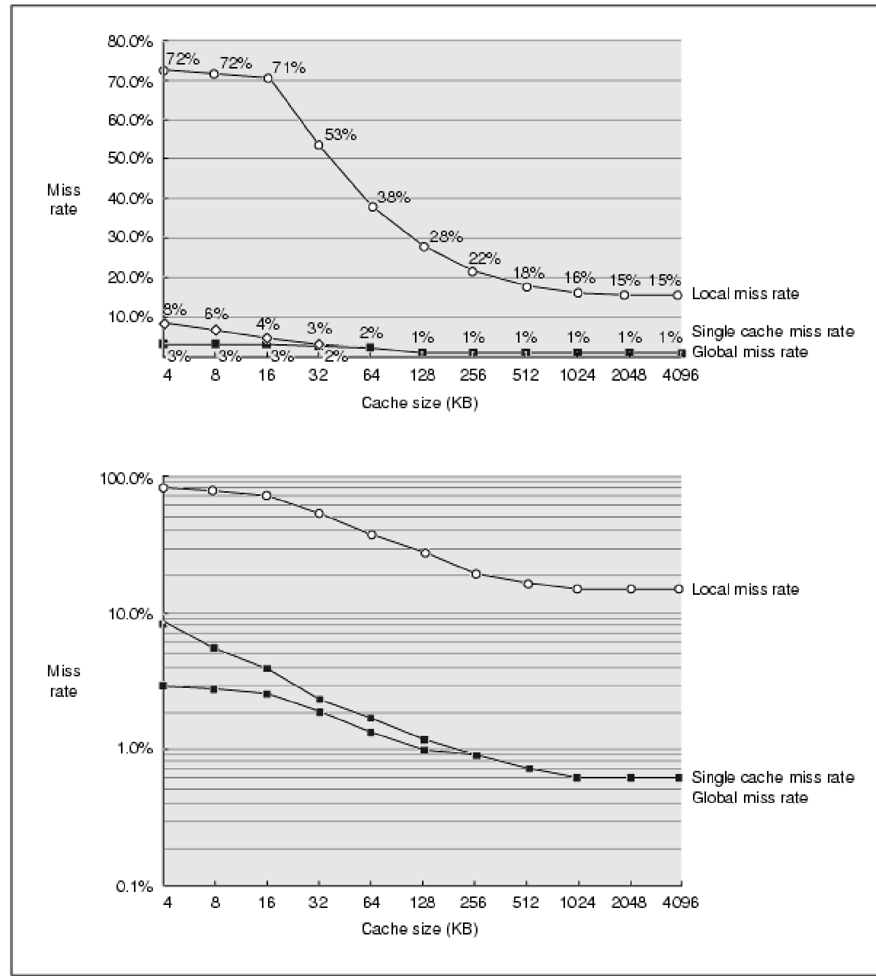
$MP_L1 = Hit_L2 + MR_L2 \times MP_L2$ SO :

$AMAT = Hit_L1 + MR_L1 \times (Hit_L2 + MR_L2 \times MP_L2)$

Where MR_L2 and MR_L1 are the 2 *Local Miss Rates:*
I.e. The number of misses in that cache divided by the total number of accesses to THAT cache only

MR_L1 x MR_L2 is the *Global Miss Rate*: I.e. The number of misses in the cache hierarchy divided by the total number of accesses generated by the CPU.

Second level caches



32KB

1st level

Second level caches' characteristics

1. The global miss rate is very similar to the single cache miss rate of the second level cache, provided that the 2nd level cache is much larger than the 1st level one.So we have the miss rate of the large 2nd level cache and the speed of the small 1st level one.
2. Local cache rate is NOT a good measure for 2nd level caches since it is a function of the Miss rate of the 1st level cache.

Second level caches' associativity

Example :

- 2-way S.A. increases hit-time by 10% of CPU clock cycle
- Hit_L2 for direct mapped = 10 clock cycles
- Local MR_L2 for direct mapped= 25%
- Local MR_L2 for 2-way S.A. = 20%
- MP_L2 = 50 clock cycles

What's one is better ?

Improving 2nd level caches

Higher associativity or pseudo-associativity are good candidates since they slightly increase the hit-time of L2. Although the large L2 eliminates the *conflict* misses by having more blocks, it also eliminates the *capacity* misses so the percentage of conflict misses is still significant.

Improving 2nd level caches

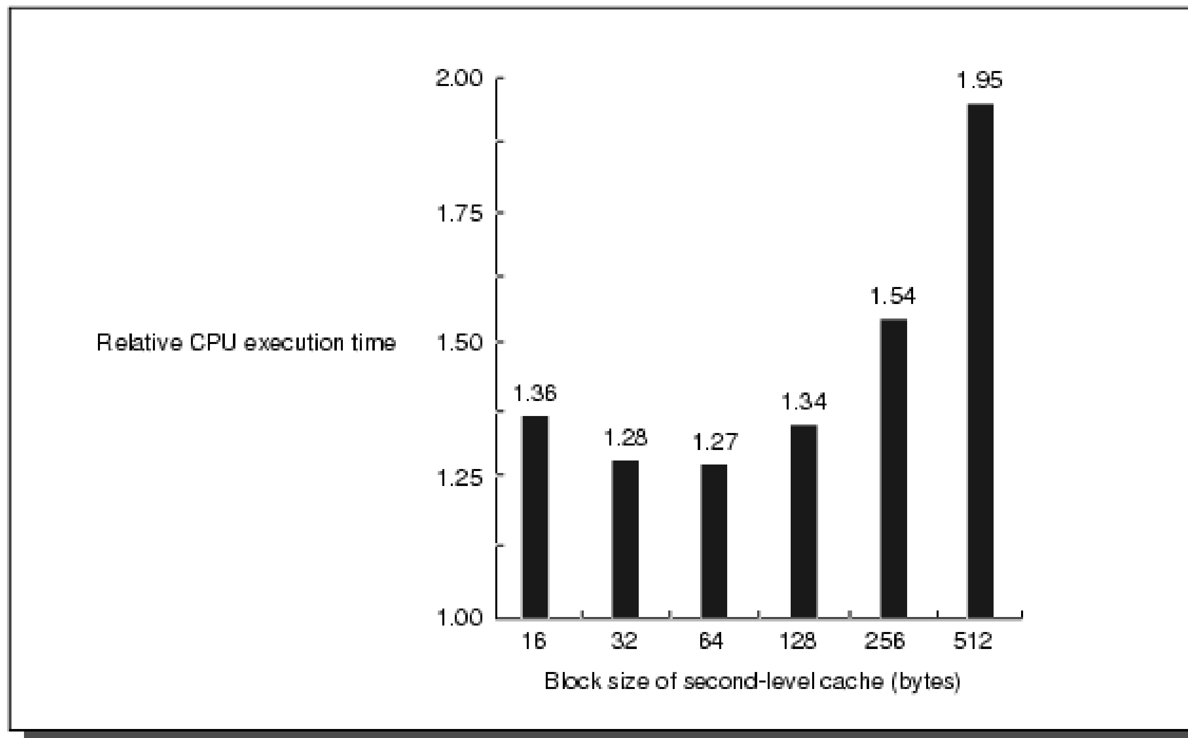
If everything in L1 are also in L2 then L2 has : *Multilevel inclusion property.* (Good in DMA and MultiProcessors) :

The problem with this property is that the two caches may have different block sizes so what about an L2 miss?

The L2 should invalidate all the block in L1 that are contained in the large L2 block which is fetched from MEM So this can cause higher L1 miss rate and higher L1 miss Penalty.

Improving 2nd level caches

Also we can increase the blocks since L2s are large and can hold a lot of blocks anyway.



Improving 2nd level caches - Summary

Since in L2 there are many fewer accesses than in L1 we are trying to have fewer misses even if that increases slightly the hit time.
As a result for L2s we have :

- Large Caches
- High Associative
- Large blocks

Reducing Hit Time: Small & Simple

Direct Mapped Caches are simpler and thus faster. They can overlap the tag check with the transmission of the data

Small caches can be put on chip (even large ones can now ☺) and … the smaller the faster .

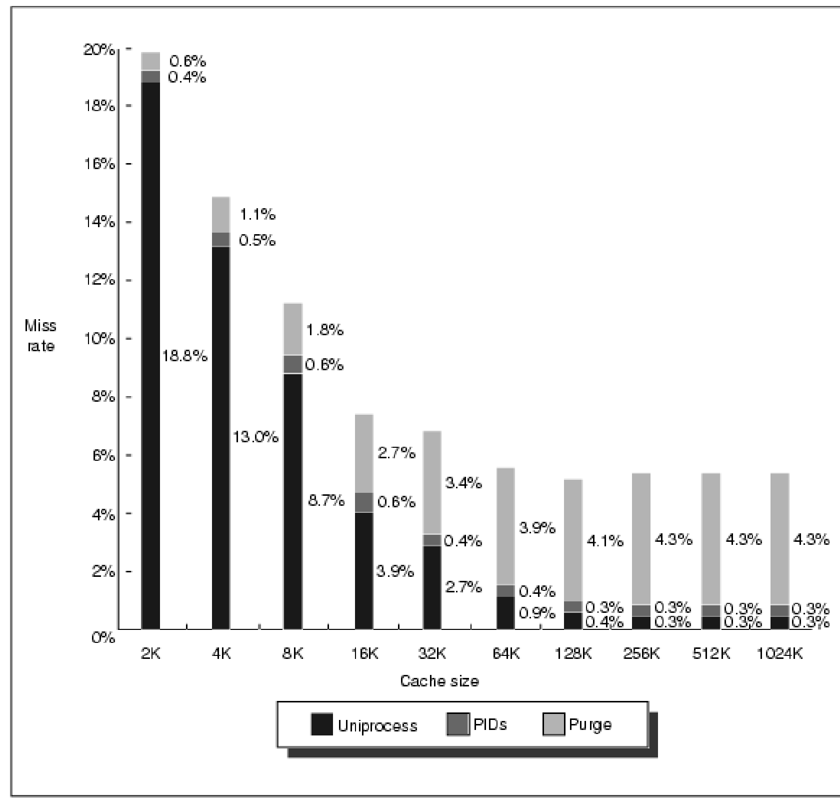
Reducing Hit Time: Avoid addr translation

CPU generates Virtual addresses so we should translate them in order to get the physical addresses used in main memory. This adds delay so why not use *Virtual Caches* (use Virtual Addresses only !) :

1. When a process changes we should flush the Cache
2. 2 virtual addresses may refer to the same physical one (One by the OS and one by the User), so the 2 *synonyms or aliases* will both be in the cache so if we modify one the other will get the wrong value!
3. I/O is using only physical address so it needs mapping to virtual Address in order to talk to a Virtual cache

Reducing Hit Time: Avoid addr translation

In order to avoid the process switching problem we can use a new field called *Process-identifier tag (PID)*



Reducing Hit Time: Avoid addr translation

Pipelining !

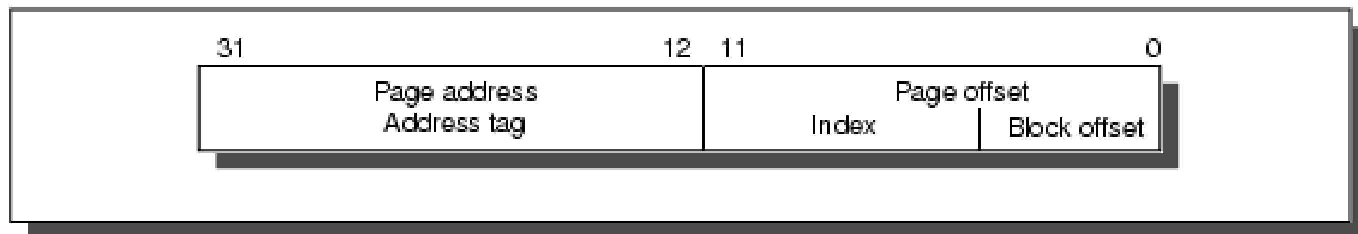
1st stage : Address Translation

2nd stage : Cache Access

This gives a fast clock cycle but you need 2 pipeline stages for the memory accesses. It complicates the hazards.

Reducing Hit Time: Avoid addr translation

Use page offset (which is the same for Physical and Virtual addresses) for indexing the cache and do in parallel the tag reading and the address translation.
BUT : In a direct-Mapped cache the maximum size of it is the size of the page. The size can be increased by increasing the Associativity : IBM 3033 uses 16-way S.A so as to have 64KB caches and 4-KB virtual pages!



Reducing Hit Time: Avoid addr translation

SOLUTIONS :

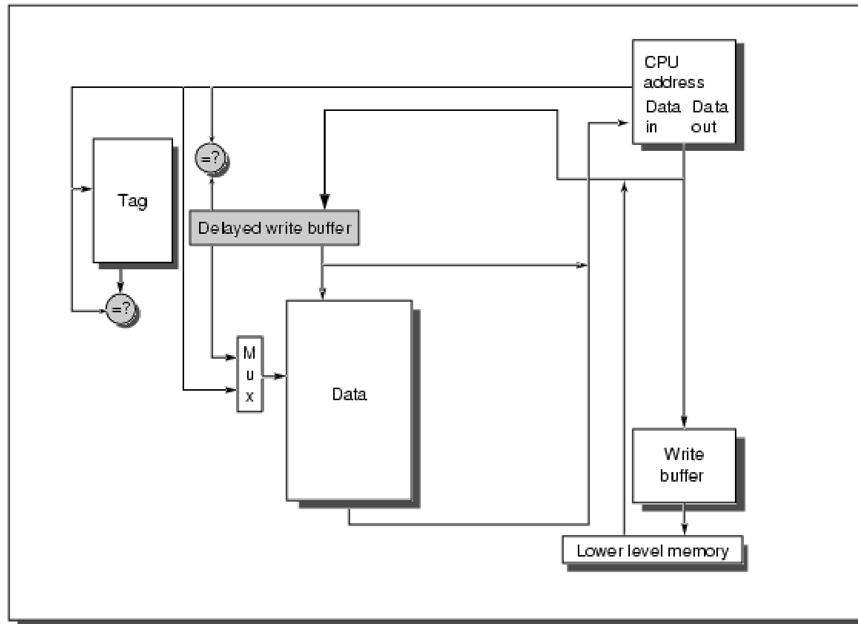
1. OS can guarantee that the last few bits of the the Virtual and Physical Addresses are always the same
2. Have a small piece of hardware that tries to guess the last bits of the physical address given the Virtual Address. If the guess is wrong we assume a pseudo Miss and then we do the actual translation and we retry .

Reducing Hit Time: Pipelining Writes

Pipelines are slower since we should first check the tag and then do the actual writing. So use a two stage pipelining :

1st stage : Compare tag with write address

2nd stage : Do the PREVIOUS writing (so 2 writes can be done back to back)



Cache Optimization Summary

Technique	Miss rate	Miss penalty	Hit time	Hardware complexity	Comment
Larger block size	+	–		0	Trivial; RS/6000 550 uses 128
Higher associativity	+		–	1	e.g., MIPS R10000 is 4-way
Victim caches	+			2	Similar technique in HP 7200
Pseudo-associative caches	+			2	Used in L2 of MIPS R10000
Hardware prefetching of instructions and data	+			2	Data are harder to prefetch; tried in a few machines; Alpha 21064
Compiler-controlled prefetching	+			3	Needs nonblocking cache too; several machines support it
Compiler techniques to reduce cache misses	+			0	Software is challenge; some machines give compiler option
Giving priority to read misses over writes		+		1	Trivial for uniprocessor, and widely used
Subblock placement		+		1	Used primarily to reduce tags
Early restart and critical word first		+		2	Used in MIPS R10000, IBM 620
Nonblocking caches		+		3	Used in Alpha 21064, R10000
Second-level caches		+		2	Costly hardware; harder if block size $L1 \neq L2$; widely used
Small and simple caches	–		+	0	Trivial; widely used
Avoiding address translation during indexing of the cache			+	2	Trivial if small cache; used in Alpha 21064
Pipelining writes for fast write hits			+	1	Used in Alpha 21064

Main Memory

Memory latency is expressed in terms of :

1. **Access time** : Time between a read is requested and a word arrives.
2. **Cycle time** : Minimum time between requests.

Cycle time > Access time since the Address lines should be stable between accesses

DRAMs and SRAMs

In DRAM we have Row (*Row Address Strobe or RAS*) and Column Addresses (*Column Address Strobe or CAS*) so as to save some pins.

Also DRAM needs *refresh* every some time (I.e. 8ms)

And when reading them, the data should be written back after the read otherwise they are destroyed ! Thus

Cycle Time > Access Time

SRAMs don’t need refreshing or writing back of data read so

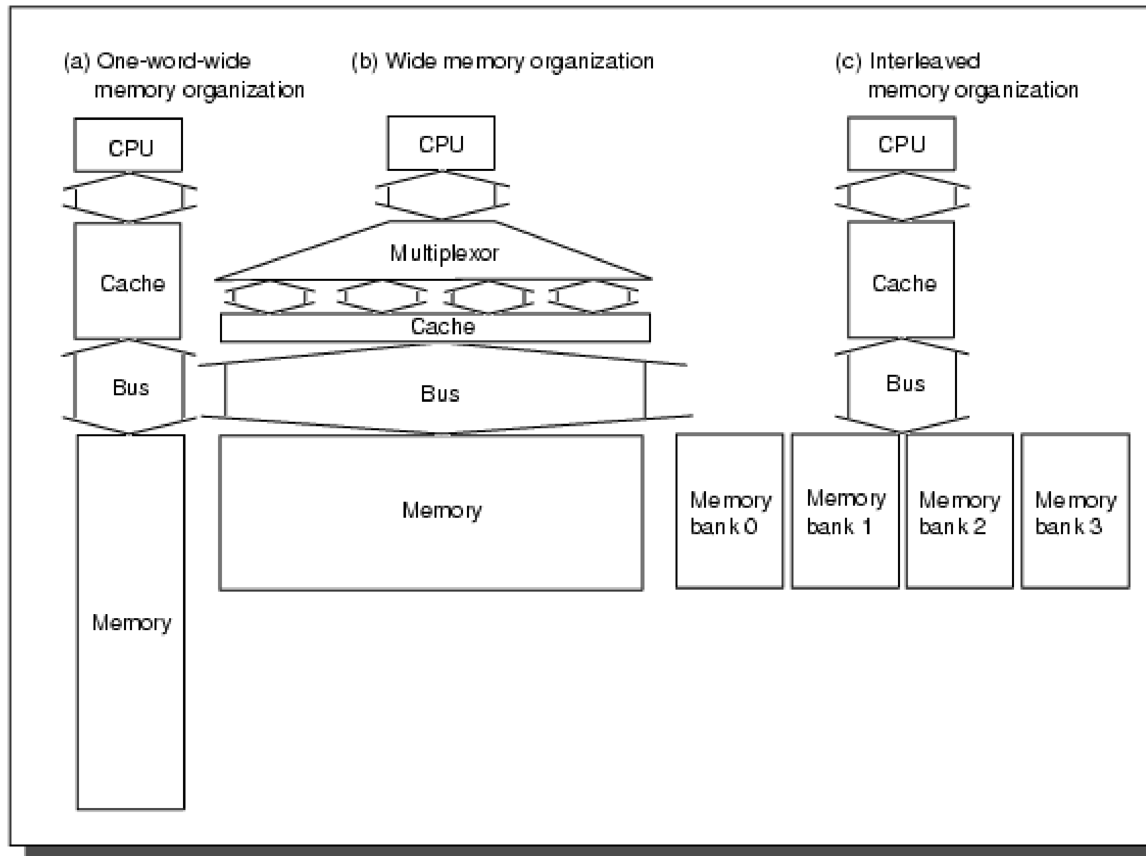
Cycle Time = Access Time

DRAMs and SRAMs

Capacity of DRAMs is 4 to 8 times larger that of SRAM
And SRAMs is 8 to 16 times faster than DRAM and
8 to 16 times more expensive … So we use DRAMs !

		Row access strobe (RAS)			
Year of introduction	Chip size	Slowest DRAM	Fastest DRAM	Column access strobe (CAS)	Cycle time
1980	64 Kbit	180 ns	150 ns	75 ns	250 ns
1983	256 Kbit	150 ns	120 ns	50 ns	220 ns
1986	1 Mbit	120 ns	100 ns	25 ns	190 ns
1989	4 Mbit	100 ns	80 ns	20 ns	165 ns
1992	16 Mbit	80 ns	60 ns	15 ns	120 ns
1995	64 Mbit	65 ns	50 ns	10 ns	90 ns

Improving Main Memory Performance



4 Cycles for address,24 Cycles access time,4 Cycles for sending a word

Wider Main Memory

You get much more bandwidth BUT :

1. CPUs access a word at a time so we need a multiplexor between the cache and the CPU which is on the critical path. 2nd level caches help with that.
2. The minimum increment in custom expandable Memories is multiplied by the width of the memory.
3. Memories with error correction have some problems when you write only a part of the line since they have to read the rest of the line and calculate the new CRC.

Simple Interleaved Memory

Send the address to all the banks but read just one at a time.

The *interleaving factor* refers to the mapping of addresses to the banks. Most commonly these memories are word-Interleaved so bank 0 has all the words whose address modulo Number-of-banks is 0 etc.

This interleaving optimizes sequential accesses.

On a cache read miss words are read sequentially.

On write back caches we also read/write blocks sequentially.

Number-of-banks >= Number of Cycles to access a word

Independent Memory Banks

Independent banks with own address lines are useful When the memory is access by more than one devices (i.e. multiprocessors) or when it wants to satisfy more than one read misses at the same time (i.e. non-blocking caches).

It's expensive and thus not used in everyday machines.

Avoiding Memory Bank Conflicts

If we have sequential accesses everything is easy. BUT what if the consecutive addresses differ by an even number? Even with 128 banks (like the NEC SX/3 machine) we will have a problem with the following code :

```
int x[256][128];
    for (j = 0; j < 512; j++)
        for (k = 0; k < 256; k++)
            x[k][j] = 2 * x[k][j];
```

Since 512 is an even multiple of 128 all the elements of a column will be in the same bank.

Avoiding Memory Bank Conflicts

SOLUTIONS :

1. Loop interchange
2. Use Prime number of banks, Since:

   Bank number = Address MOD Number of Banks

(2) Address within bank = Address / Number of Banks

And according to the *Chinese Remainder Theorem*

if the number of banks is one less a power of two and the number of words in a bank a power of two (always!),

(2)_can be written as :

Address within bank = Address MOD Number of Banks

MOD is **fast** in hardware!

DRAM-Specific Interleaving

DRAMs have rows and columns and different columns within the same row can be accessed quickly(We buffer a whole row every time in a fast buffer . The row size is the square root of the DRAM size so :

16Kb for a 64Mb DRAM and 64Kb for a 256Mb DRAM

- *<u>Nibble mode</u>*: DRAM can supply 3 extra bits from sequential locations for every RAS.
- *<u>Page mode</u>*: But changing column address we can access any bit of the row until a new row access(or refresh) arrives
- *<u>Static column</u>*: Page mode but without strobing the CAS

DRAM-Specific Interleaving (2)

Chip size	Row access		Column access	Cycle time	Optimized time nibble, page, static column
	Slowest DRAM	Fastest DRAM			
64 Kbits	180 ns	150 ns	75 ns	250 ns	150 ns
256 Kbits	150 ns	120 ns	50 ns	220 ns	100 ns
1 Mbits	120 ns	100 ns	25 ns	190 ns	50 ns
4 Mbits	100 ns	80 ns	20 ns	165 ns	40 ns
16 Mbits	80 ns	60 ns	15 ns	120 ns	30 ns
64 Mbits	65 ns	50 ns	10 ns	90 ns	25 ns

Optimized time is the same no matter which of the 3 optimized modes is selected.
So with nibble mode for example we can get 4 bits externally in the time of 4 optimized cycles, very useful for a 4-way interleaved memory.

RAMBUS/VRAMs interfaces

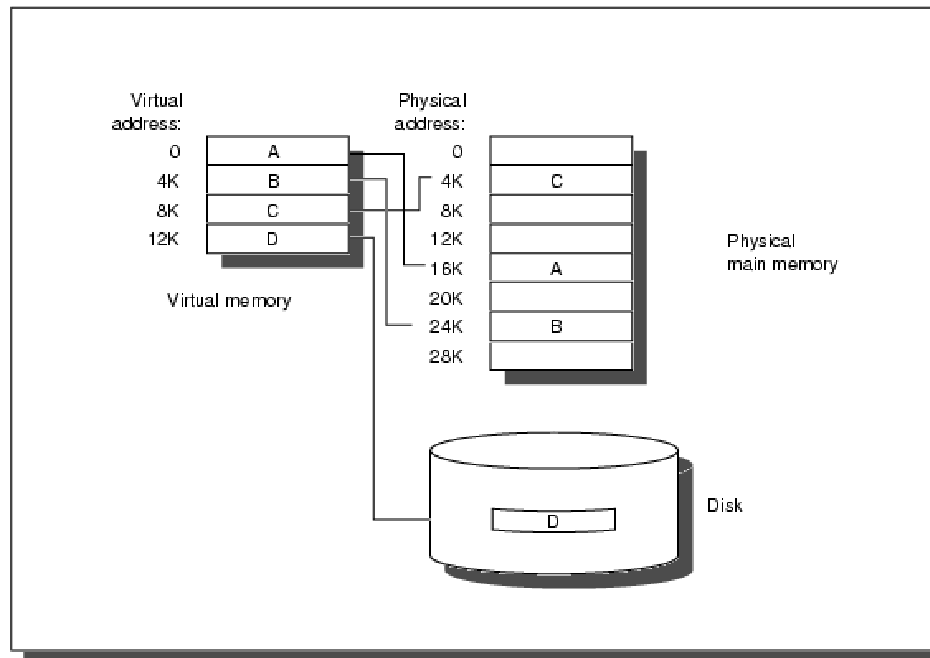
Instead of RAS/CAS we have a bus that allows other accesses over it between the sending of the address and the receiving of the data. A chip can return a variable amount of Data from a single request, it has a byte-wide interface and a Clock so as to be synchronized with the CPU. Once the address pipeline is full a chip can deliver 1 byte every **2ns!** It costs 20% more per megabyte over a standard DRAM.

VRAM is optimized so as to get more bandwidth when you have serial access (used in video cards)

Virtual Memory

Used for :

1. Sharing memory between processes
2. Extending the useful memory (instead of using *overlay*)



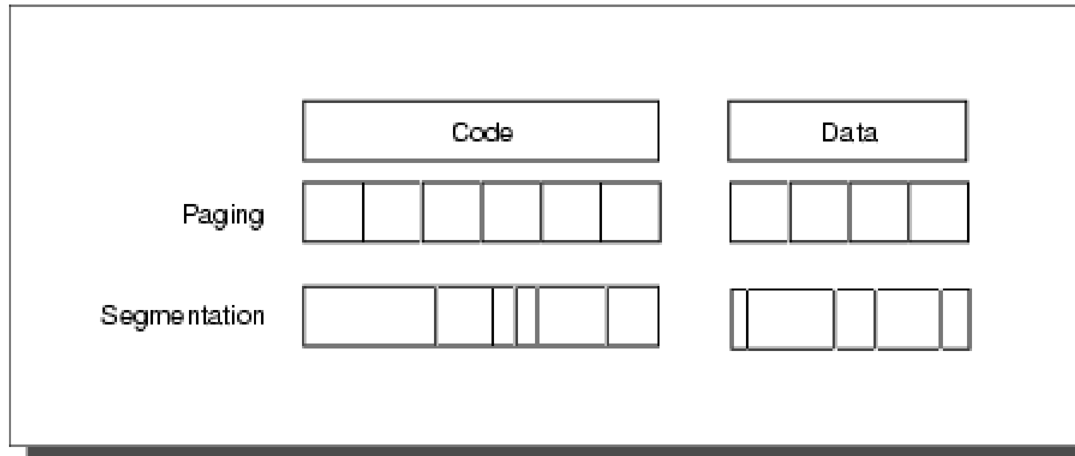
Virtual Memory and Cache

Page or Segment	vs	Block
Page or Address fault	vs	Miss
Replacement by OS	vs	Replacement by HW
Size is defined by CPU	vs	Size is arbitrary

Parameter	First-level cache	Virtual memory
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–2 clock cycles	40–100 clock cycles
Miss penalty	8–100 clock cycles	700,000–6,000,000 clock cycles
(Access time)	(6–60 clock cycles)	(500,000–4,000,000 clock cycles)
(Transfer time)	(2–40 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.5–10%	0.00001– 0.001%
Data memory size	0.016–1MB	16–8192 MB

Pages and Segments

Pages are fixed length while Segments are variable length.



	Page	Segment
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Hard (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

Virtual Memory questions

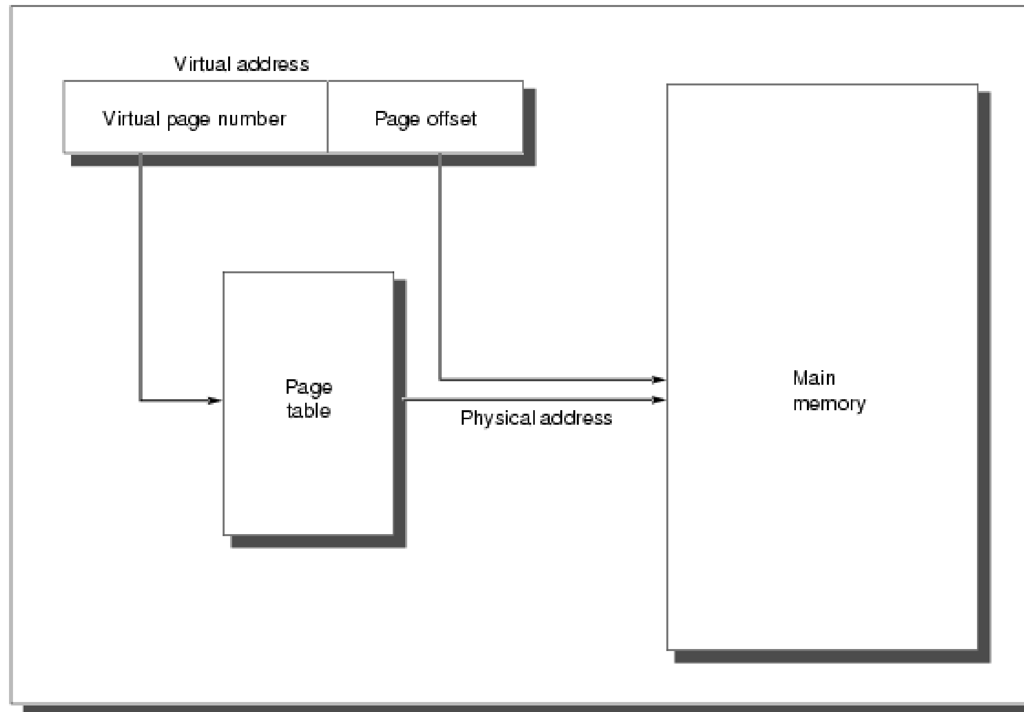
- **Block placement**
  - Where can a block be placed ?
  - Anywhere since miss penalty is huge so miss rate should be minimal (fully associative).
- **Block identification**
  - How is a block found if it is in the Main memory?
  - Using a page table which is indexed by the virtual addr
- **Block replacement**
  - Which block should be replaced on a miss?
  - LRU algorithm by OS. The OS is using a *reference bit*
- **Write Strategy**
  - What happens on a write?
  - Write back since it takes ages to write to the hard disk

Address Translation

Page tables are large so they are stored in Main Memory

So each access takes at least twice as long :

1 cycle to take the physical address and 1 to get the data

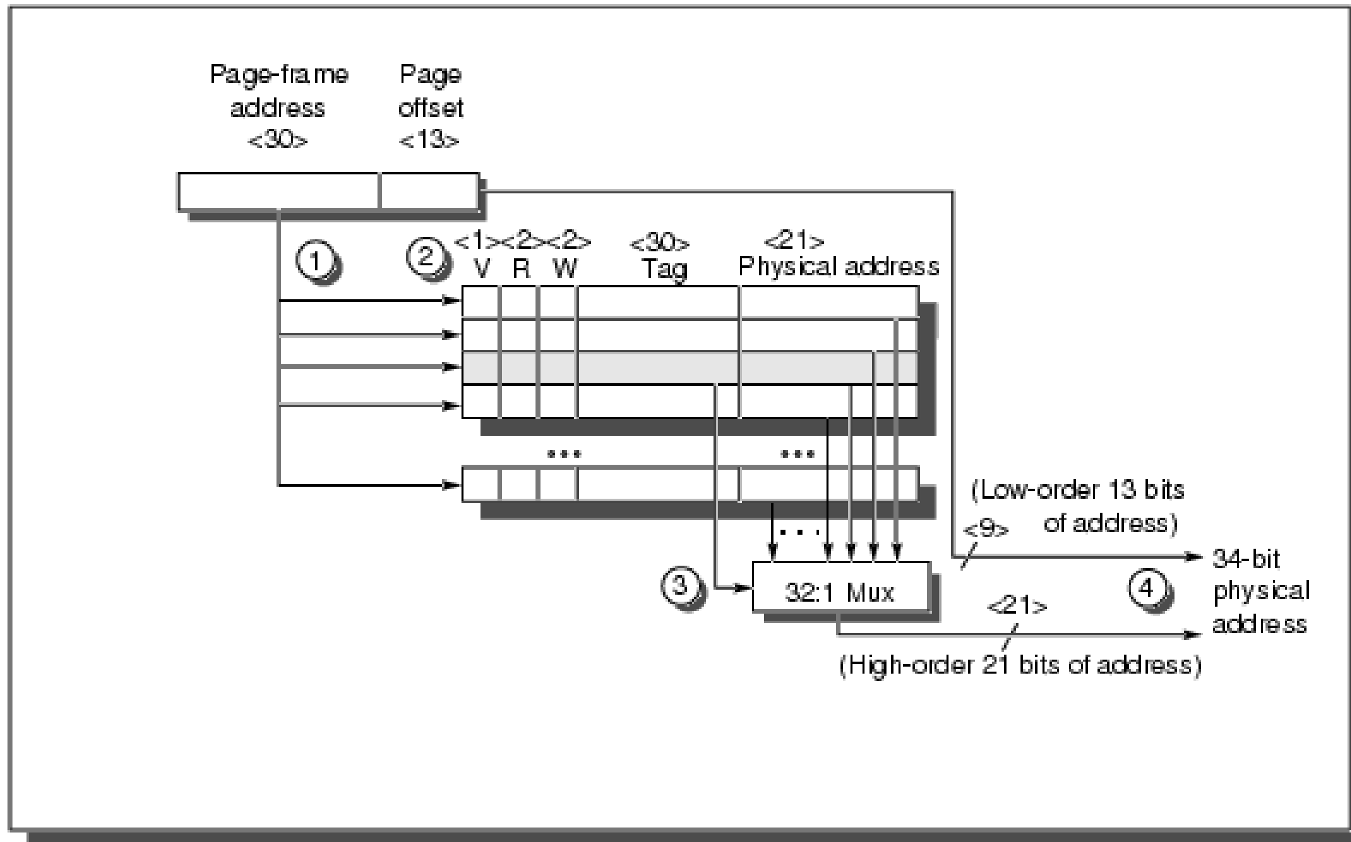


Fast Address Translation, TLBs

If accesses have locality so should addresses :
So we use a memory address special cache where we put the most recently used address translations, and as a result we rarely have to do two memory accesses.
This is called a *Translation look-aside buffer* or **TLB** .

In the TLB we store the Virtual - Physical Address pairs and some control bits like *valid/dirty/use/read/write*.

Alpha 21064's TLB



Tag is the Virtual Address

No need to translate the page offset since it is the same in both addresses.

TLB with Caches

The CPU generates virtual addresses so :

1. If cache is small we use only the page offset so the cache can be accessed without address translation. We need the physical address only for comparison with the cache tag

2. If cache is large then the TLB is on the critical path so it should be fast. So it is always small (32 block is typical), and in sometimes is it even pipelined.

Selecting a page size

Large pages are preferable since :

1. The size of the page table is inversely proportional to the page size, so larger pages result in smaller page tables.

2. A large page can simplify the cache accesses (no translation needed if size of the cache < page size).

3. Transferring larger pages is more efficient.

4. The TLB misses are smaller when the pages are larger and thus fewer.

Selecting a page size (2)

Small pages are not that bad either when storage is limited

When we have small pages less storage is wasted when a continuous region of virtual memory is not equal in size to a multiple of the page size. This is called *internal fragmentation* . The average wasted storage per process is 1.5 times the page size so this is negligible for machines with 4KB or 8KB pages. It may be important for machines with small main memory and pages of 64KB or more.

Also large pages would increase the time for starting-up a small process since a large page should be read and only a small part of it needed.

Crosscutting Issues in Designing Mem Systems

- Instruction Level Parallelism versus Reducing Cache Misses

```
for (k = 0; k < 512; k++)
    for (j = 1; j < 512; j++)
        x[k][j] = 2 * x[k][j-1];
```

OR :

```
for (k = 0; k < 512; k++)
     for (j = 1; j < 512; j+=4){
        x[k][j] = 2 * x[k][j-1];
        x[k][j+1] = 2 * x[k][j];
        x[k][j+2] = 2 * x[k][j+1];
        x[k][j+3] = 2 * x[k][j+2];}
```

We have low cache miss rate (access data the way they are stored) BUT we have RAW dependencies.

Crosscutting Issues in Designing Mem Systems

By interchanging the two loops and unrolling we get :

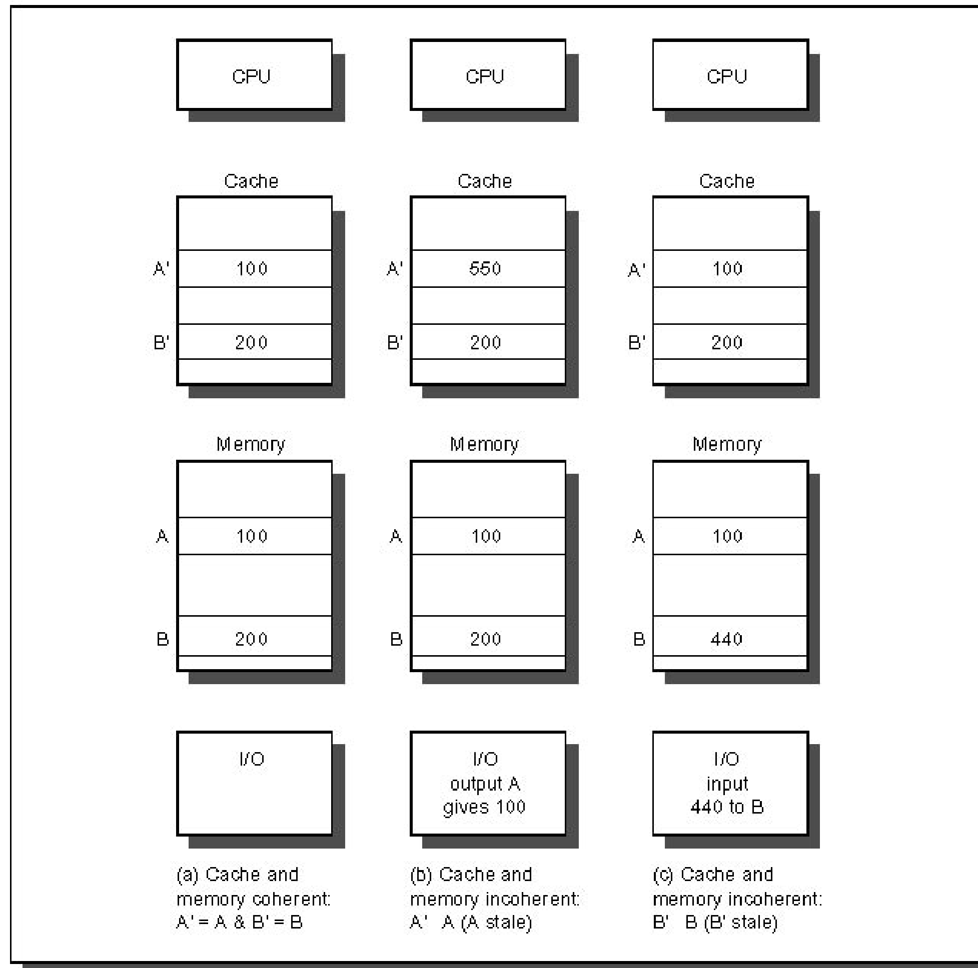
```
for (j = 1; j < 512; j++)
            for (k = 0; k < 512; k++)
                  x[k][j] = 2 * x[k][j-1];
```

OR :

```
      for (j = 1; j < 512; j++)
             for (k = 0; k < 512; k+=4){
                  x[k][j] = 2 * x[k][j-1];
                  x[k+1][j] = 2 * x[k+1][j];
                  x[k+2][j] = 2 * x[k+2][j+1];
                  x[k+3][j] = 2 * x[k+3][j+2]; }
```

We have no RAW (all statements are independent) BUT possibly a much higher miss rate !

Cache coherency problem



Crosscutting Issues in Designing Mem Systems

•I/O and Consistency of Cached Data (*Cache-coherency*)

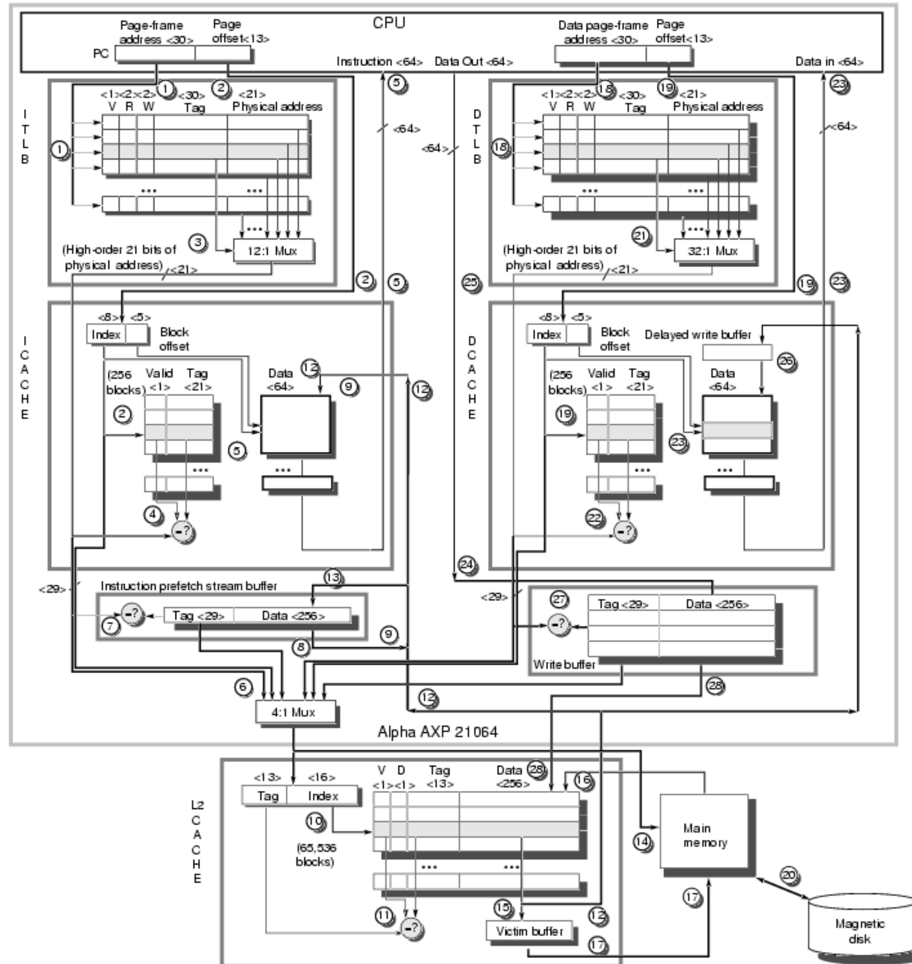
Where does the I/O occur ?

Between the I/O and the cache **OR** the memory ?

If the I/O talk directly to the **cache** the CPU and the I/O see the same data **BUT** the CPU should wait when the I/O downloads data, we put useless data in the small cache and we put more h/w in the critical cache access path.

If the I/O talks to the **memory** and we have a write-through cache there is no problem for outputs. For inputs, no blocks written by the I/O should be in the cache. So either a page should be defined as non-cachable or all the blocks to be written by the I/O should be Flushed from the cache after the input occurs.

Alpha 21064 Memory Hierarchy



Alpha 21064 Performance

Program	CPI			Total cache	Instr. issue	Other stalls	Total CPI	Miss rates		
	I cache	D cache	L2					I cache	D cache	L2
TPC-B (db1)	0.57	0.53	0.74	1.84	0.79	1.67	4.30	8.10%	41.00%	7.40%
TPC-B (db2)	0.58	0.48	0.75	1.81	0.76	1.73	4.30	8.30%	34.00%	6.20%
AlphaSort	0.09	0.24	0.50	0.83	0.70	1.28	2.81	1.30%	22.00%	17.40%
Avg comm	0.41	0.42	0.66	1.49	0.75	1.56	3.80	5.90%	32.33%	10.33%
espresso	0.06	0.13	0.01	0.20	0.74	0.57	1.51	0.84%	9.00%	0.33%
li	0.14	0.17	0.00	0.31	0.75	0.96	2.02	2.04%	9.00%	0.21%
eqntott	0.02	0.16	0.01	0.19	0.79	0.41	1.39	0.22%	11.00%	0.55%
compress	0.03	0.30	0.04	0.37	0.77	0.52	1.66	0.48%	20.00%	1.19%
sc	0.20	0.18	0.04	0.42	0.78	0.85	2.05	2.79%	12.00%	0.93%
gcc	0.33	0.25	0.02	0.60	0.77	1.14	2.51	4.67%	17.00%	0.46%
Avg SPECint92	0.13	0.20	0.02	0.35	0.77	0.74	1.86	1.84%	13.00%	0.61%
spice	0.01	0.68	0.02	0.71	0.83	0.99	2.53	0.21%	36.00%	0.43%
doduc	0.16	0.26	0.00	0.42	0.77	1.58	2.77	2.30%	14.00%	0.11%
mdljdp2	0.00	0.31	0.01	0.32	0.83	2.18	3.33	0.06%	28.00%	0.21%
wave5	0.04	0.39	0.04	0.47	0.68	0.84	1.99	0.57%	24.00%	0.89%
tomcatv	0.00	0.42	0.04	0.46	0.67	0.79	1.92	0.06%	20.00%	0.89%
ora	0.00	0.10	0.00	0.10	0.72	1.25	2.07	0.05%	7.00%	0.10%
alvinn	0.03	0.49	0.00	0.52	0.62	0.25	1.39	0.38%	18.00%	0.01%
ear	0.01	0.15	0.00	0.16	0.65	0.24	1.05	0.11%	9.00%	0.01%
mdljsp2	0.00	0.09	0.00	0.09	0.80	1.67	2.56	0.05%	5.00%	0.11%
swm256	0.00	0.24	0.01	0.25	0.68	0.37	1.30	0.02%	13.00%	0.32%
su2cor	0.03	0.74	0.01	0.78	0.66	0.71	2.15	0.41%	43.00%	0.16%
hydro2d	0.01	0.54	0.01	0.56	0.69	1.23	2.48	0.09%	32.00%	0.32%
nasa7	0.01	0.68	0.02	0.71	0.68	0.64	2.03	0.19%	37.00%	0.25%
fpppp	0.52	0.17	0.00	0.69	0.70	0.97	2.36	7.42%	7.00%	0.01%
Avg SPECfp92	0.06	0.38	0.01	0.45	0.71	0.98	2.14	0.85%	20.93%	0.27%

Memory hierarchies Summary

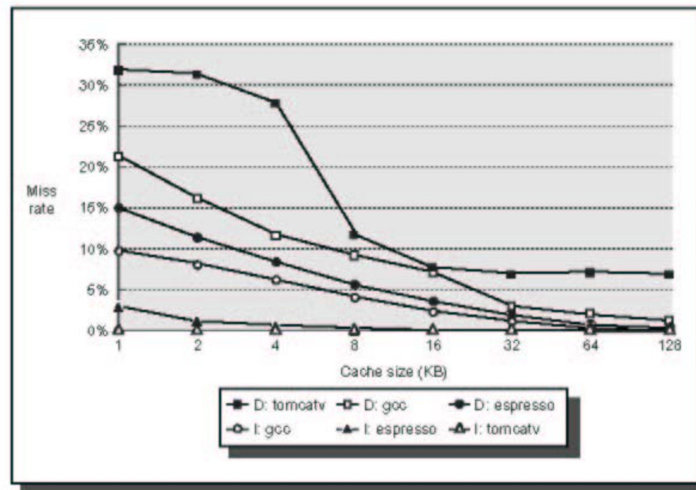
	TLB	First-level cache	Second-level cache	Virtual memory
Block size	4–8 bytes (1 PTE)	4–32 bytes	32–256 bytes	4096–16,384 bytes
Hit time	1 clock cycle	1–2 clock cycles	6–15 clock cycles	10–100 clock cycles
Miss penalty	10–30 clock cycles	8–66 clock cycles	30–200 clock cycles	700,000–6,000,000 clock cycles
Miss rate (local)	0.1–2%	0.5–20%	15–30%	0.00001–0.001%
Size	32–8192 bytes (8–1024 PTEs)	1–128 KB	256 KB–16 MB	16–8192 MB
Backing store	First-level cache	Second-level cache	Page-mode DRAM	Disks
Q1: block placement	Fully associative or set associative	Direct mapped	Direct mapped or set associative	Fully associative
Q2: block identification	Tag/block	Tag/block	Tag/block	Table
Q3: block replacement	Random	N.A. (direct mapped)	Random	≈ LRU
Q4: write strategy	Flush on a write to page table	Write through or write back	Write back	Write back

Fallacies and Pitfalls

Pitfall: Too small an address space.

PDP-11, 8080, 8086, 80186, 80286, Zilog Z80, Cray 1 all died of few address bits. It's too difficult to add address bits since it determines the minimum length of Program Counter, memory accesses, address arithmetic.

Fallacy:Predicting cache performance of one program from another.



Fallacies and Pitfalls

Pitfall: Ignoring the impact of the OS on the performance of the Memory hierarchy.

Workload	Misses		Time						
			% time due to appl. misses		% time due directly to OS misses				% time OS misses & appl. conflicts
	% in appl	% in OS	Inherent appl. misses	OS conflicts w. appl.	OS instr misses	Data misses for migration	Data misses in block operations	Rest of OS misses	
Pmake	47%	53%	14.1%	4.8%	10.9%	1.0%	6.2%	2.9%	25.8%
Multipgm	53%	47%	21.6%	3.4%	9.2%	4.2%	4.7%	3.4%	24.9%
Oracle	73%	27%	25.7%	10.2%	10.6%	2.6%	0.6%	2.8%	26.8%

Fallacies and Pitfalls

Pitfall: Simulating enough instructions to get accurate performance measures of the memory hierarchy.
1) Use a small trace for predicting the behavior of a large cach
2) Locality is not constant over the run of the entire program