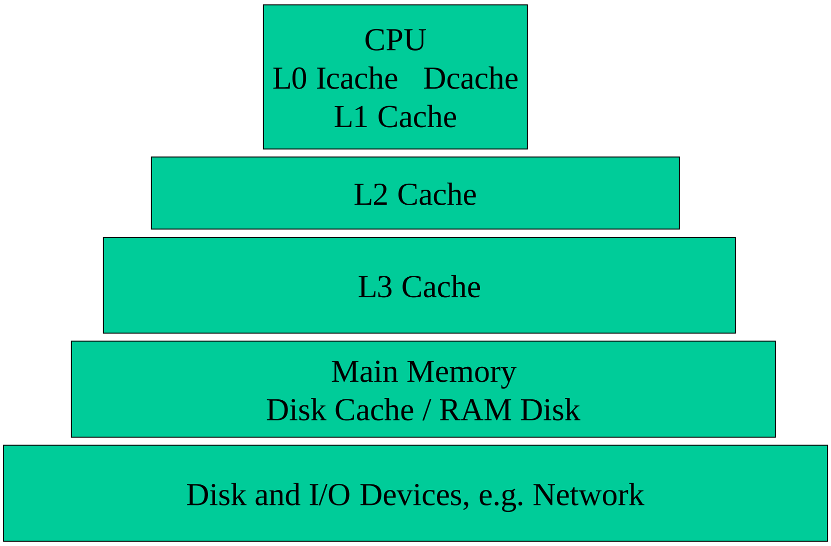
Caching Review and Performance

Memory Hierarchies

- Takes advantage of **locality of reference principle**
  - Most programs do not access all code and data uniformly, but repeat for certain data choices
    - spatial – nearby references are likely
    - temporal – repeat reference is likely
  - Fast memory is expensive per byte
  - Make a hierarchy with fast at the top (but not much memory) and slow at the bottom (but lots of memory)

Sample Memory Hierarchy



CPU
L0 Icache Dcache
L1 Cache

L2 Cache

L3 Cache

Main Memory
Disk Cache / RAM Disk

Disk and I/O Devices, e.g. Network

Hierarchy Concepts

- Data access in terms of **blocks** at each level
  - Size of a block varies, especially from L1 to L2, etc.
  - Hit : data you want is found in the cache
  - Miss: data you want is not found in the cache, must be fetched from the lower level memory system
    - Misses cause stall cycles
    - Stall Cycles = IC * Mem-Refs-Per-Instruction * Miss Rate * Miss Penalty
- Four major questions regarding any level in the hierarchy
  - Q1 : Where can a block be placed in the upper level?
  - Q2 : How is a block found in the upper level?
  - Q3 : Which block should be replaced on a miss?
  - Q4 : What happens on a write?

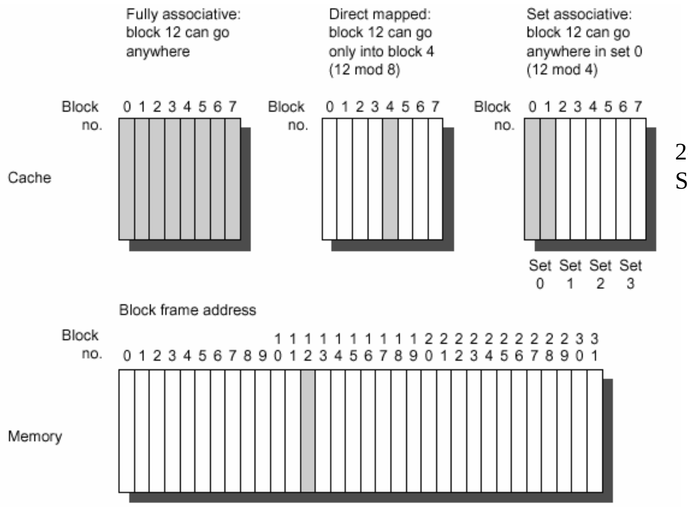
Cache Parameters

- Some typical parameters for caches
- Block / Line Size
  - 16 to 256 bytes
- Hit time
  - 1 Cycle in L1 Cache
- Miss penalty
  - 10 to hundreds of clock cycles
- Access time of next lower memory
  - 4 - 32 clock cycles
- Miss rate
  - 1% to 20% (application dependent)
- L1 Cache Size
  - 64-640K

Cache Strategies – Block Placement

- Direct Mapped
  - Each block has only one place to go in a cache, typically
    - Address mod Num-Blocks-In-Cache
    - Usually lower n bits corresponds to the offset in the block, where 2^n = Block size, and then another m bits corresponding to the cache block, where 2^m = Num blocks in the cache
- Fully Associative
  - Block can be placed anywhere in the cache
  - Implies we must be able to search for it associatively
- Set Associative
  - A **set** is a group of arbitrary blocks in the cache
  - An address in memory maps to a set, then maps into a block within that set
  - Usually lower n bits corresponds to the offset in the block, where 2^n = Block size, and then another m bits corresponding to the set, where 2^m = Num sets;

Block Placement



2-way
Set associative

Block Identification

- Physical Address
  - For direct-mapped:
    - Tag ## Block-Index ## Block-Offset
  - For set-associative
    - Tag ## Set-Index ## Block-Offset

Tag	Set	Offset
r bits	m bits	n bits

In set-associative cache

2^m sets possible

r bits searched simultaneously for matching block

2^n gives offset to the data into the matching block

–For fully associative, just tag and offset

Typical Cache

- May be composed of 2 types of fast memory devices
  - SRAM's - hold the actual data and address and status tags in a direct mapped cache
  - TAG RAM's help with the accounting for set-associative cache
- TAG RAM's are small associative memories
  - provide fully parallel search capability
  - sequential search would take too long so not even an option
- Where do you have to search
  - fully associative - everywhere
  - set associative - only within the set
  - direct mapped - no search
    - just check for valid and compare one ID

Finding a Block

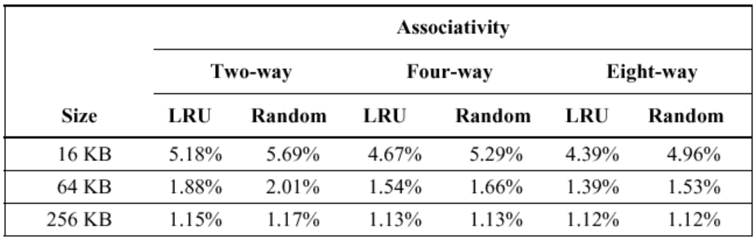
- We've already covered how to look up a block in either scheme
  - Find block via direct map or associative mapping, perhaps first finding the right set and then comparing the tag bits for a match
  - Go to the offset of this block to get the memory data
- Extra details
  - Valid bit
    - Added to indicate if a tag entry contains a valid address
  - Dirty bit
    - Added to indicate if data has been written to

Block Replacement

- Random
  - – Pick a block at random and discard it
  - – For set associative, randomly pick a block within the mapped set
  - – Sometimes use pseudo-random instead to get reproducibility at debug time
- LRU - least recently used
  - – Need to keep time since each block was last accessed
  - – Expensive if number of blocks is large due to global compare
  - – Approximation is often used; Use bit tag or counter and LFU
- Replacement strategy critical for small caches
  - – doesn't make a lot of difference for large ones
  - – ideal would be a least-likely prediction scheme
  - – No simple scheme known for least-likely prediction

Miss Rates

- On benchmark traces, block size of 16 bytes



	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%

- Not much difference with larger caches
- Not much difference with eight-way scheme

Cache Writes

- Reads dominate cache access
  - – Only 7% of overall memory traffic are writes on RISC
  - – Implies we should make reads fastest (which is good because it is easier to handle)
    - Can read block the same time we compare the tag for a match; ignore value if there is no match
    - Can’t do this with writes, if we compare the tag and write at the same time, we’d have to undo the write if there was no match
      - – So we wait an extra cycle for result of the tag comparison
      - – Complicated more if we write multiple bytes

Cache Write Schemes

- Write Through
  - – Write information back to the cache and to the lower-level memory at the same time
    - Lower level access is slower, though
  - – Maintains memory consistency for other devices
    - Other processors
    - DMA
- Write Back
  - – Write to cache only, but set the dirty bit when we write
  - – Dirty blocks are written back to memory only when replaced
  - – Faster, independent of main memory speeds
    - But causes complications with memory consistency

Write Stalls

- Occur when we must stall for a write to complete
- Write miss may generate write stall
  - Write Buffers allows processor to act as soon as data written to the buffer, providing overlapping execution with memory
    - decrease write stalls but do not eliminate them
- Common operations on a write miss
  - write allocate
    - load the block, do the write
    - usually the choice for write back caches so the data is available for a subsequent read or writes
  - No-write allocate or write around
    - modify the block in the lower-level memory, do not load the block in the cache
    - usually the choice for write through caches since subsequent writes would go to memory anyway

Improving Cache Performance

- Processor needs data and instructions
- Two separate caches often implemented
  - Avoids structural hazard problems with fetching instruction, data we discussed with pipelining
  - The two caches have separate access patterns
- When there is one cache containing both, it is called a unified or mixed cache

Cache Performance

- Average memory access time = Hit time + (Miss Rate * Miss Penalty)
- Memory Stall Clock Cycles = (Reads * Read Miss Rate * Read Penalty) + (Writes * Write Miss Rate * Write Penalty)

- Sometimes we combine reads and writes as an approximation using a generic “Miss”
  - Memory Stall Clock Cycles = Memory Accesses * Miss Rate * Miss Penalty

- CPUTime = IC * (CPI$_{exec}$ + Mem Accesses/Instr * Miss Rate * Miss Penalty) * Clock Cycle Time

Cache Performance Example

Cache Miss Penalty = 50 cycles
Instructions normally take 2 cycles, ignoring stalls
Cache Miss rate is 2%
Average of 1.33 Memory References per instruction

Impact of cache vs. no cache?

CPUTime = IC * (CPI$_{exec}$ + Mem Accesses/Instr * Miss Rate * Miss Penalty) * Clock Cycle Time

CPUTime(cache) = IC * (2 + 1.33 * 0.02 * 50) * Cycle Time
= 3.33 * IC * Cycle Time ; from 2 to 3.33 when not perfect
CPUTime(nocache) = IC * (2 + 1.33 * 50) * Cycle Time
= 68.5 ; Over 30 times longer!

Cache Performance Limitations

- Caches can have a huge impact on performance
- Downside
  - The lower the CPI(exec) the higher the relative impact of a fixed number of cache miss clock cycles
  - CPU's with higher clock rates and same memory hierarchy has a larger number of clock cycles per miss
- Bottom line : Amdahl's Law strikes again, impact of caching can slow us down as we get high clock rates
- Set-Associative Cache appears to perform best on the simulation data
  - Implementing set-associative cache requires some extra multiplexing to select the block we want in the set
  - Increases basic cycle time the larger each set is
  - This basic cycle time could make set-associative caches slower than direct-mapped caches in some cases!

Sources of Cache Misses – 3 C's

- Compulsory
  - first access to a block, no choice but to load it
  - also called cold-start or first-reference misses
- Capacity
  - if working set is too big for the cache then even after steady state they all won't fit
  - Therefore, needed lines will be displaced by other needed lines
  - thrashing possible if we later want to retrieve something tossed
- Conflict
  - Collision as a result of the block placement strategy
  - Data you want maps to the same block in the cache
- Data on these three miss types for 1-8 way caches

Miss Rate Per Type SPEC2000

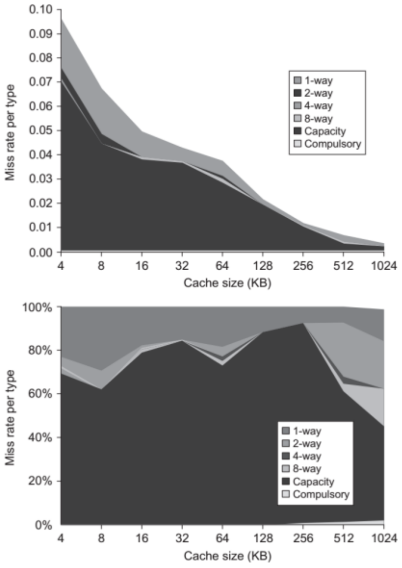


Figure B.9 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to the three C's for the data in Figure B.8. The top diagram shows the actual data cache miss rates, while the bottom diagram shows the percentage in each category. (*Space allows* the graphs to show one extra cache size than can fit in Figure B.8.)

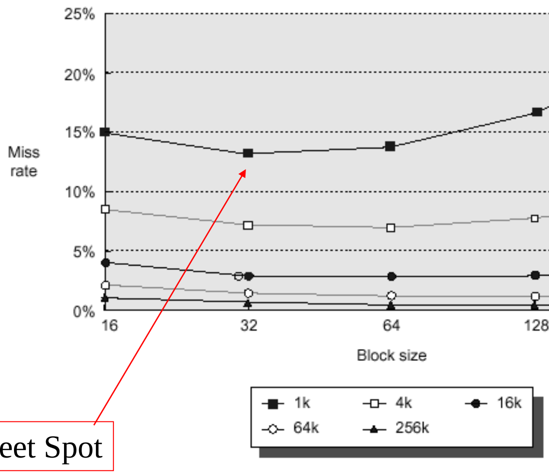
Reducing Cache Misses

- We'll examine seven techniques
  - – Larger Block Size
  - – Higher Associativity
  - – Victim Caches
  - – Pseudo-Associative Caches
  - – Hardware prefetching
  - – Compiler prefetching
  - – Compiler Optimizations

#1: Increased Block Size

- Advantages
  - Reduce compulsory misses due to spatial locality
- Disadvantages
  - Larger block takes longer to move, so higher penalty for miss
  - More conflicts now though, because there are fewer blocks in the cache, so more memory blocks map to the same cache blocks

Miss Rate vs. Block Size



Sweet Spot

Cache Size

#2: Higher Associativity

- 2:1 Rule of Thumb
  - 2 way set associative cache of size N/ 2 is about the same as a direct mapped cache of size N
  - So does this mean even more associations is better?
- Advantage
  - Higher associativity should reduce conflicts
- Disadvantage
  - Higher associativity can reduce number of sets, if we keep the same cache size
  - There is overhead with higher associativity in the hardware, increases the basic clock cycle for all instructions

Associativity Example

Assume higher associativity increases the clock cycle as:
CycleTime(2-way) = 1.10 * CycleTime(1-way)
CycleTime(4-way) = 1.12 * CycleTime(1-way)
CycleTime(8-way) = 1.14 * CycleTime(1-way)

A hit takes 1 cycle, miss penalty for direct-map is 50 cycles
Calculate Ave. Mem Access Times

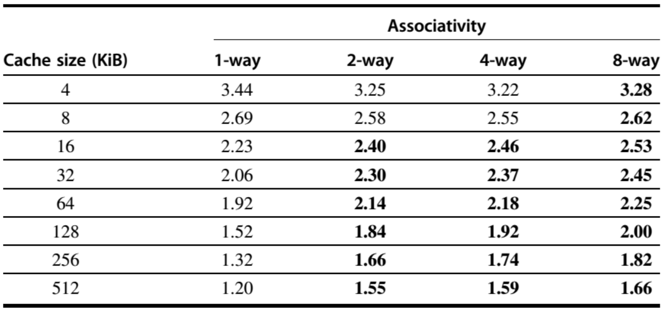
Ave mem Access Time = HitTime + miss Rate * Miss Penalty

AveTime(1-way) = 1 + MissRate(1-way) * 50
AveTime(2-way) = 1.10 * MissRate(2-way) * 50
…

(continued on next slide)

Associativity Example (2)

Look up miss rates for the different caches from the previous table
e.g. for direct-mapped cache, miss rate = 0.133 at 1Kb cache
AveTime(1-way) = 1 + 0.133 * 50 = 7.65
…
Gives us the table below:



	Associativity			
Cache size (KiB)	**1-way**	**2-way**	**4-way**	**8-way**
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. *Boldface* type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

#3: Victim Caches

- Idea: with direct mapping, conflict only occurs with a small number of blocks
  - Might occur frequently, but with not too many blocks
  - Very bad if we thrash among these direct mapped blocks to the same cache block
  - So use a small, fully-associative cache to store what was thrown out
- Add a small "victim" cache between the main cache and main memory
  - This cache only stores data discarded from a cache miss
  - Keep it small, so it is easy to implement, even associatively
  - If data is not in the main cache but is in the victim cache, swap the data from the main/victim cache
  - Since we address the victim cache and main cache at the same time, there is no increased penalty with this scheme

Victim Caches

- Study by Jouppi
  - Victim cache of 1-5 entries effective at reducing conflict misses for small, direct-mapped caches
  - Removed 20-95% of conflict misses in a 4K direct mapped cache
    - Of course this is a very small cache by today’s standards
    - Not as much benefit with larger caches, even if direct-mapped, due to alleviation of conflicts

29

#4: Pseudo-Associative Cache

- Also called column associative
- Idea
  - start with a direct mapped cache, then on a miss check another entry
    - A typical next location to check is to invert the high order index bit to get the next try
  - Similar to double hashing
- Initial hit fast (direct), second hit slower
  - may have the problem that you mostly need the slow hit
  - in this case it’s better to swap the blocks
  - like victim caches - provides selective on demand associativity

30

#5: Hardware Prefetch

- Get proactive!
- Modify our hardware to prefetch into the cache instructions and data we are likely to use
  - Alpha AXP 21064 fetches two blocks on a miss from the I-cache
    - Requested block and the next consecutive block
    - Consecutive block catches 15-25% of misses on a 4K direct mapped cache, can improve with fetching multiple blocks
  - Similar approach on data accesses not so good, however
- Works well if we have extra memory bandwidth that is unused
- Not so good if the prefetch slows down instructions trying to get to memory

#6 Compiler-Controlled Prefetch

- Two types
  - Register prefetch (load value into a register)
  - Cache prefetch (load data into cache, need new instr)
- The compiler determines where to place these instructions, ideally in such a way as to be invisible to the execution of the program
  - Nonfaulting instructions – if there is a fault, the instruction just turns into a NOP; just a hint and speculative
- Only makes sense if cache can continue to supply data while waiting for prefetch to complete
  - Called a *nonblocking* or *lockup-free* cache
- Loops are a key target

Compiler Prefetch Example

Using a Write-Back cache

```
for (i=0; i<3; i++)
   for (j=0; j<100; j++)
       a[i][j]=b[j][0]+b[j+1][0];
```

Spatial locality
Say even j's miss, odd hit
Total of 300/2 = 150 misses

Temporal locality
Misses on i=0
Misses on i=0,j=0 only
Total of 101 misses

Prefetched version, assuming we need to prefetch 7 iterations in advance to avoid the miss penalty. Doesn't address initial misses:

```
for (j=0; j<100; j++) {
   prefetch(b[j+7][0]);
   prefetch(a[0][j+7]);
   a[0][j]=b[j][0]+b[j+1][0];
}
for (i=1; i<3; i++)
   for (j=0; j<100; j++) {
       prefetch(a[i][j+7]);
        a[i][j]=b[j][0]+b[j+1][0]; }
```

Fetch for 7 iterations later
Pay penalty for first 7 iterations

Total misses = (3*7/2) + 1 + 7
= 19

#7 Compiler Optimizations

- Lots of options
- Array merging
  - allocate arrays so that paired operands show up in same cache block
- Loop interchange
  - exchange inner and outer loop order to improve cache performance
- Loop fusion
  - for independent loops accessing the same data
  - fuse these loops into a single aggregate loop
- Blocking
  - Do as much as possible on a sub-block before moving on

Array Merging

Given a loop like this:

```
int val1[SIZE], val2[SIZE];
for (i=0; i<1000; i++) {
    x += val1[i] * val2[i];
}
```

For spatial locality, instead use:

```
struct merge {
   int val1, val2;
} m[SIZE];

for (i=0; i<1000; i++) {
    x += m[i].val1 * m[i].val2;
}
```

For some situations, array splitting is better:

```
struct merge {
   int val1, val2;
} m1[SIZE], m2[SIZE];

for (i=0; i<1000; i++) {
  x += m1[i].val1 * m2[i].val1;
}
```

val2 unused, getting in the way of spatial locality. First version could actually be better!

Objects can be good or bad, depending on access pattern

Loop Interchange

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

Say the cache is small, much less than 5000 numbers
We'll have many misses in the inner loop due to replacement

Switch order:

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

With spatial locality, presumably we can operate on all 100 items in the inner loop without a cache miss

Access all words in the cache block before going on to the next one

Loop Fusion

```
for (i=0; i<100; i++) {
   for (j=0; j< 5000; j++)
       a[i][j]=1/b[i][j] * c[i][j];
}
for (i=0; i<100; i++) {
   for (j=0; j< 5000; j++)
       d[i][j]=a[i][j] * c[i][j];
}
```

Merge loops:

```
for (i=0; i<100; i++) {

   for (j=0; j< 5000; j++)

       a[i][j]=1/b[i][j] * c[i][j];
       d[i][j]=a[i][j] * c[i][j];

}
```

Freeload on cached value!

Reducing Miss Penalties

- So far we've been talking about ways to reduce cache misses
- Let's discuss now reducing access time (the penalty) when we have a miss
- What we've seen so far
  - #1: Write Buffer
    - Most useful with write-through cache
    - no need for the CPU to wait on a write
      - hence buffer the write and let the CPU proceed
      - needs to be associative so it can respond to a read of a buffered value

Problems with Write Buffers

- Consider this code sequence
  - SW R3, 512(R0) ← Maps to cache index 0
  - LW R1, 1024(R0) ← Maps to cache index 0
  - LW R2, 512(R0) ← Maps to cache index 0
- There is a Read After Write (RAW) data hazard
  - Store is put into write buffer
  - First load puts data from M[1024] into cache index 0
  - Second load results in a miss, if the write buffer isn't done writing, the read of M[512] could put the old value in the cache and then R2
- Solutions
  - Make the read wait for write to finish
  - Check the write buffer for contents first

39

#2 Other Ways to Reduce Miss Penalties

- Sub-Block Placement
  - Large blocks reduce tag storage and increase spatial locality, but more collisions and a higher penalty in transferring big chunks of data
  - Compromise is Sub-Blocks
  - Add a "valid" bit to units smaller than the full block, called sub-blocks
    - Allow a single sub-block to be read on a miss to reduce transfer time
    - In other modes of operation, we fetch a regular-sized block to get the benefits of more spatial locality

40

#3 Early Restart & Critical Word First

- CPU often needs just one word of a block at a time
  - Idea : Don’t wait for full block to load, just pass on the requested word to the CPU and finish filling up the block while the CPU processes the data
- Early Start
  - As soon as the requested word of the block arrives, send it to the CPU
- Critical Word First
  - Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling in the rest of the block

#4 Nonblocking Caches

- Scoreboarding or Tomasulo-based machines
  - Could continue executing something else while waiting on a cache miss
  - This requires the CPU to continue fetching instructions or data while the cache retrieves the block from memory
  - Called a nonblocking or lockup-free cache
  - Cache could actually lower the miss penalty if it can overlap multiple misses and combine multiple memory accesses

#5 Second Level Caches

- Probably the best miss-penalty reduction technique, but does throw in a few extra complications on the analysis side…
- L1 = Level 1 cache, L2 = Level 2 cache

$$Average\_Memory\_Access\_Time = Hit\_Time(L1) + Miss\_Rate(L1) \times Miss\_Penalty(L1)$$

$$Miss\_Penalty(L1) = Hit\_Time(L2) + Miss\_Rate(L2) \times Miss\_Penalty(L2)$$

- Combining gives:

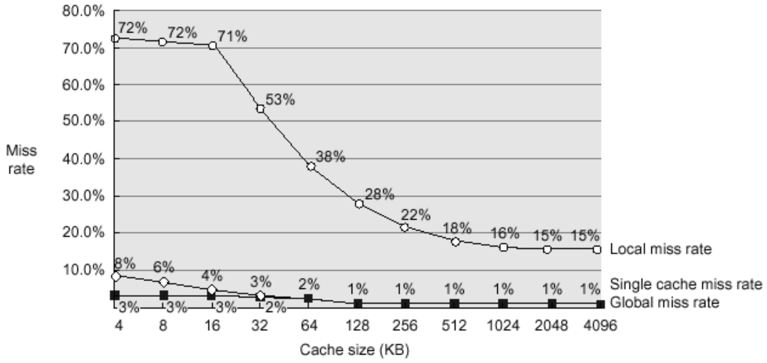
$$\begin{gathered}Average\_Memory\_Access\_Time = Hit\_Time(L1) + Miss\_Rate(L1) \times \\ \left(Hit\_Time(L2) + Miss\_Rate(L2) \times Miss\_Penalty(L2)\right)\end{gathered}$$

  - little to be done for compulsory misses and the penalty goes up
  - capacity misses in L1 end up with a significant penalty reduction since they likely will get supplied from L2
  - conflict misses in L1 will get supplied by L2 unless they also conflict in L2

Second Level Caches

- Terminology
  - Local Miss Rate
    - Number of misses in the cache divided by total accesses to the cache; this is Miss Rate(L2) for the second level cache
  - Global Miss Rate
    - Number of misses in the cache divided by the total number of memory accesses generated by the CPU; the global miss rate of the second-level cache is
      - Miss Rate(L1)*Miss Rate(L2)
    - Indicates fraction of accesses that must go all the way to memory
  - If L1 misses 40 times, L2 misses 20 times for 1000 references
    - 40/1000 = 4% local miss rate for L1
    - 20/40 = 50% local miss rate for L2
    - 20/40 * 40/1000 = 2% = global miss rate for L2

Effects of L2 Cache



L2 cache with 32K L1 cache
Top: local miss rate of L2 cache
Middle: L1 cache miss rate
Bottom: Global miss rate

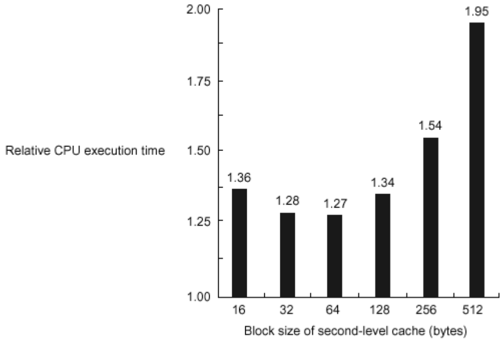
Takeaways:
Size of L2 should be > L1
Local miss rate not a good measure

Size of L2?

- L2 should be bigger than L1
  - Everything in L1 likely to be in L2
  - If L2 is just slightly bigger than L1, lots of misses
- Size matters for L2, then..
  - Could use a large direct-mapped cache
    - Large size means few capacity misses, compulsory or conflict misses possible
  - Set associativity make sense?
    - Generally not, more expensive and can increase cycle time
  - Most L2 caches made as big as possible

L2 Cache Block Size

- Increased block size
  - Big block size increases chances for conflicts (fewer blocks in the cache), but not so much a problem in L2 if it's already big to start with
  - Sizes of 64-256 bytes are popular



L2 Cache Inclusion

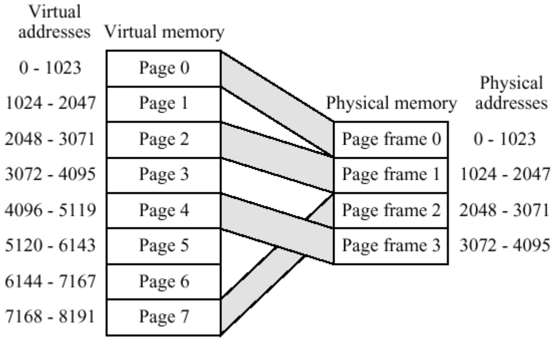
- Should data in L1 also be in L2?
  - If yes, L2 has the *multilevel inclusion property*
  - This can be desirable to maintain consistency between caches and I/O; we could just check the L2 cache
  - Write through will support multilevel inclusion
- Drawback if yes:
  - "Wasted" space in L2, since we'll have a hit in L1
  - Not a big factor if L2 >> L1
  - Write back caches
    - L2 will need to "snoop" for write activity in L1 if it wants to maintain consistency in L2

Reducing Hit Time

- We've seen ways to reduce misses, and reduce the penalty.. next is reducing the hit time
- #1 Simplest technique: Small and Simple Cache
  - Small → Faster, less to search
  - Must be small enough to fit on-chip
    - Some compromises to keep tags on chip, data off chip but not used today with the shrinking manufacturing process
  - Use direct-mapped cache
    - Choice if we want an aggressive cycle time
    - Trades off hit time for miss rate, since set-associative has a better miss rate

#2 Virtual Caches

- Virtual Memory
  - Map a virtual address to a physical address or to disk, allowing a virtual memory to be larger than physical memory
- Traditional caches or Physical caches
  - Take a physical address and look it up in the cache



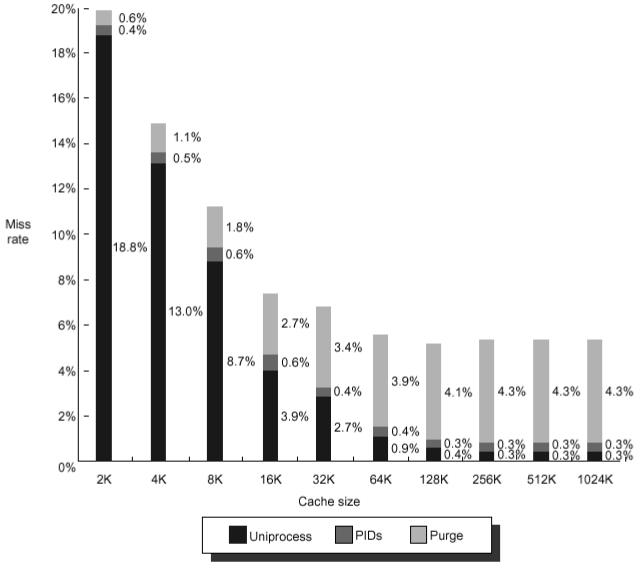
Virtual Caches

- Virtual caches
  - Same idea as physical caches, but start with the virtual address instead of the physical address
  - If data is in the cache, it avoids the costly lookup to map from a virtual address to a physical address
    - Actually, we still need to the do the translation to make sure there is no protection fault
- Too good to be true?

Virtual Cache Problems

- Process Switching
  - When a process is switched, the same virtual address from a previous process can now refer to a different physical addresses
    - Cache must be flushed
    - Too expensive to save the whole cache and re-load it
    - One solution: add PID's to the cache tag so we know what process goes with what cache entry
  - Comparison of results and the penalty on the next slide

Miss Rates of Virtually Addressed Cache



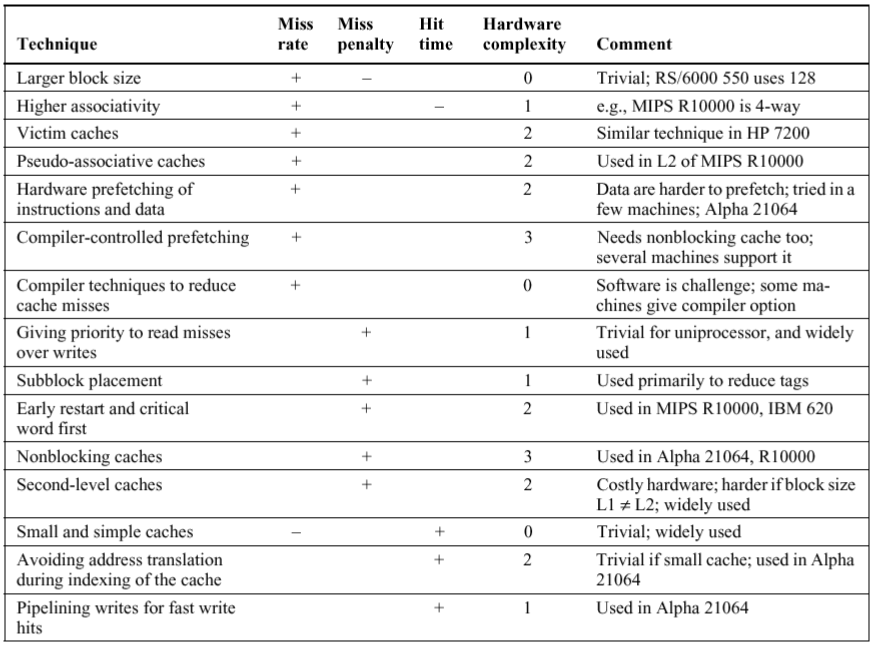
More Virtual Cache Problems…

- Aliasing
  - – Two processes might access different virtual addresses that are really the same physical address
  - – Duplicate values in the virtual cache
  - – Anti-aliasing hardware guarantees every cache block has a unique physical address
- Memory-Mapped I/O
  - – Would also need to map memory-mapped I/O devices to a virtual address to interact with them
- Despite these issues…
  - – Virtual caches used in some of today’s processors

#3 Pipelining Writes for Fast Hits

- Write hits take longer than read hits
  - Need to check the tags first before writing data to avoid writing to the wrong address
  - To speed up the process we can pipeline the writes (Alpha)
    - First, split up the tags and the data to address each independently
    - On a write, cache compares the tag with the write address
    - Writes to the data portion of the cache can occur in parallel with a comparison of some other tag
      - We just overlapped two stages
    - Allows back-to-back writes to finish one per clock cycle
- Reads play no part in this pipeline, can already operate in parallel with the tag check

Cache Improvement 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 ≠ 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