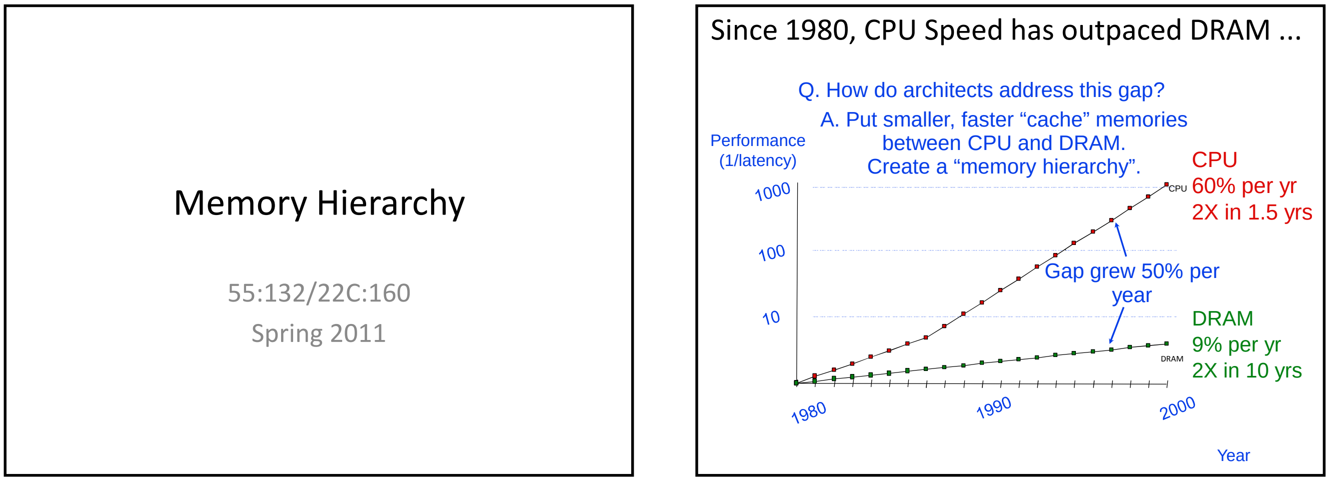
Memory Hierarchy

55:132/22C:160

Spring 2011



Since 1980, CPU Speed has outpaced DRAM ...

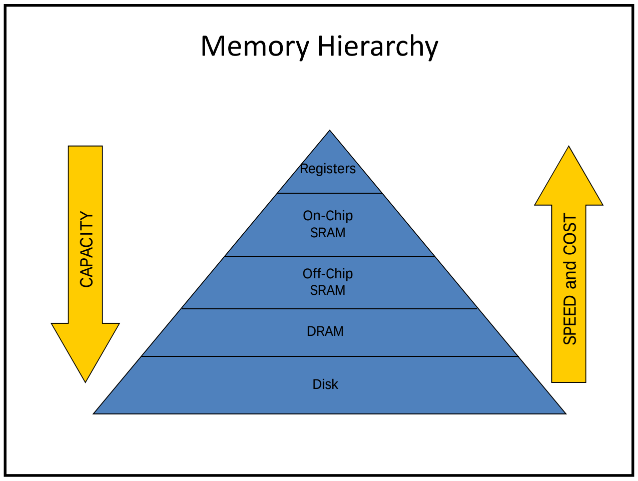
Q. How do architects address this gap?

A. Put smaller, faster "cache" memories between CPU and DRAM. Create a "memory hierarchy".

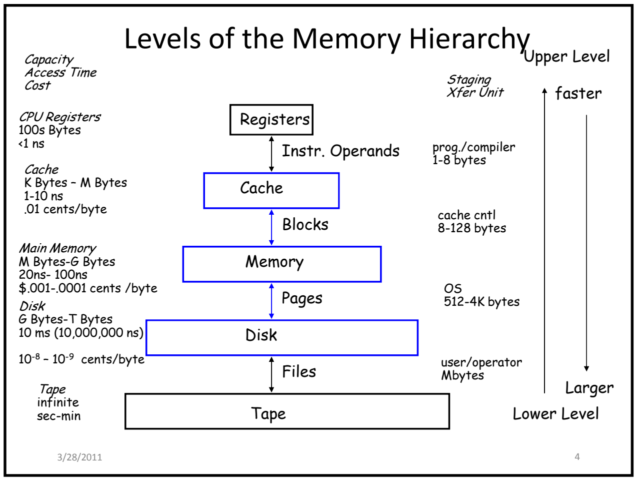
CPU
60% per yr
2X in 1.5 yrs

Gap grew 50% per year

DRAM
9% per yr
2X in 10 yrs



Memory Hierarchy



Levels of the Memory Hierarchy

Capacity
Access Time
Cost

CPU Registers
100s Bytes
<1 ns

Cache
K Bytes – M Bytes
1-10 ns
.01 cents/byte

Main Memory
M Bytes-G Bytes
20ns- 100ns
$.001-.0001 cents /byte

Disk
G Bytes-T Bytes
10 ms (10,000,000 ns)

10^{-8} – 10^{-9} cents/byte

Tape
infinite
sec-min

Staging
Xfer Unit

prog./compiler
1-8 bytes

cache cntl
8-128 bytes

OS
512-4K bytes

user/operator
Mbytes

3/28/2011 4

Why De We Need a Memory Hierarchy?

- Processors consume lots of memory bandwidth, e.g.:

$$BW = \frac{1.0 inst}{cycle} \times \left[\frac{1 Ifetch}{inst} \times \frac{4B}{Ifetch} + \frac{0.4 Dref}{inst} \times \frac{4B}{Dref} \right] \times \frac{1 Gcycles}{\mathrm{sec}}$$
$$= \frac{5.6 GB}{\mathrm{sec}}$$

- Need lots of memory
  - Gbytes to multiple TB
- Must be cheap per bit
  - (TB x anything) is a lot of money!
- These requirements seem incompatible

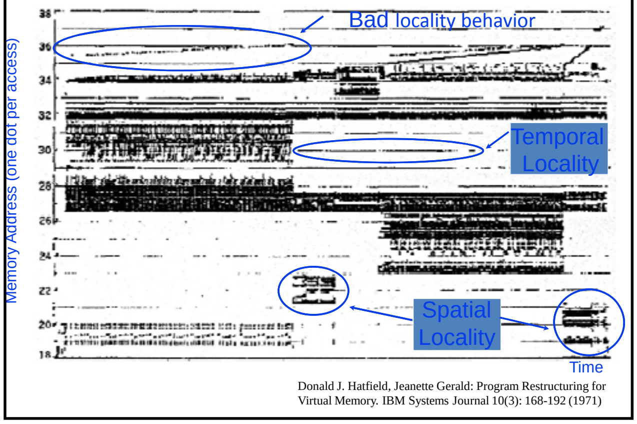
Memory Hierarchy

- Fast and small memories (SRAM)
  - Enable quick access (fast cycle time)
  - Enable lots of bandwidth (1+ Load/Store/I-fetch/cycle)
  - Expensive, power-hungry
- Slower larger memories (DRAM)
  - Capture larger share of memory
  - Still relatively fast
  - Cheaper, low-power
- Slow huge memories (DISK, SSD)
  - Really huge (Tbytes)
  - Really cheap (think $100/Tbyte
  - Really slow
- All together: provide appearance of large, fast memory with cost of cheap, slow memory

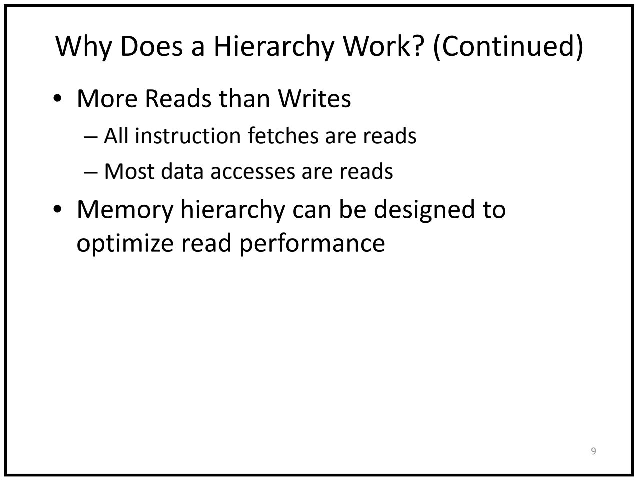
Why Does a Hierarchy Work?

- Locality of reference
  - Temporal locality
    - Reference same memory location repeatedly
  - Spatial locality
    - Reference near neighbors around the same time
- Empirically observed
  - Significant!
  - Even small local storage (8KB) often satisfies >90% of references to multi-MB data set

Programs with locality cache well ...

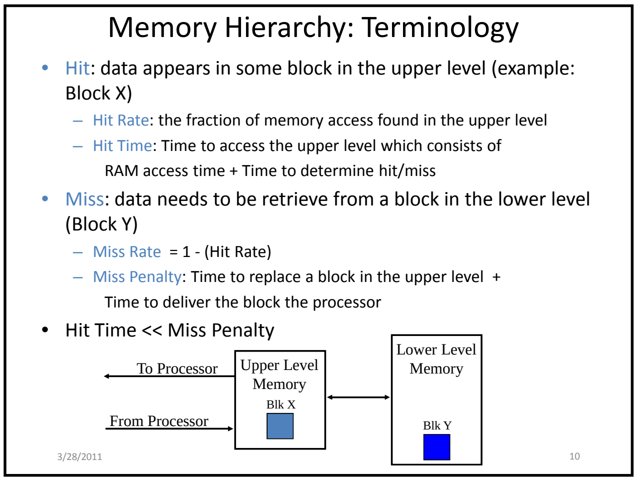


Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)



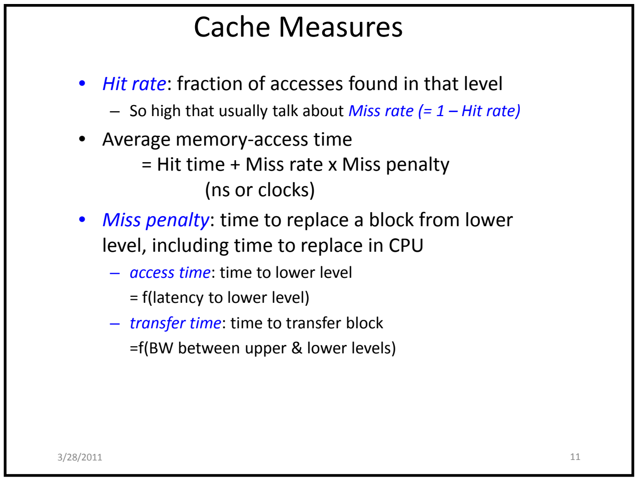
Why Does a Hierarchy Work? (Continued)

- More Reads than Writes
  - All instruction fetches are reads
  - Most data accesses are reads
- Memory hierarchy can be designed to optimize read performance



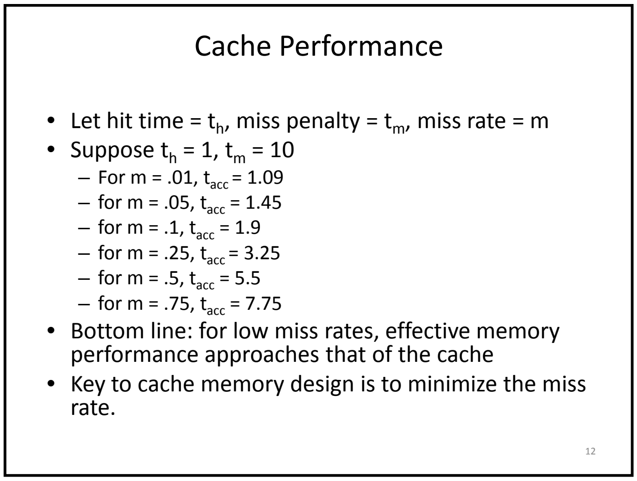
Memory Hierarchy: Terminology

- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty



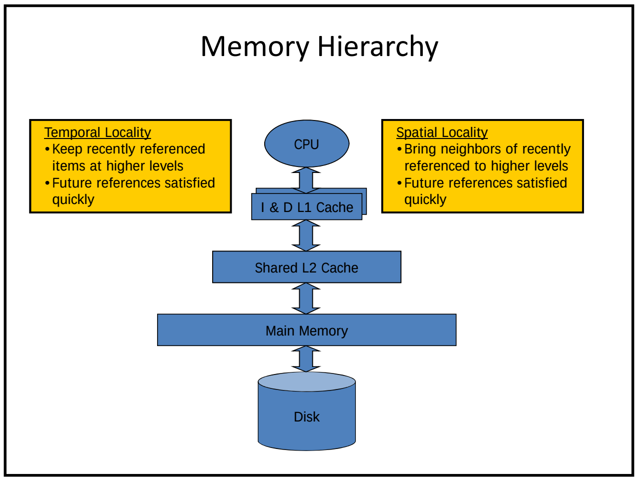
Cache Measures

- *Hit rate*: fraction of accesses found in that level
  - So high that usually talk about *Miss rate (= 1 – Hit rate)*
- Average memory-access time
  = Hit time + Miss rate x Miss penalty
  (ns or clocks)
- *Miss penalty*: time to replace a block from lower level, including time to replace in CPU
  - *access time*: time to lower level
    = f(latency to lower level)
  - *transfer time*: time to transfer block
    =f(BW between upper & lower levels)



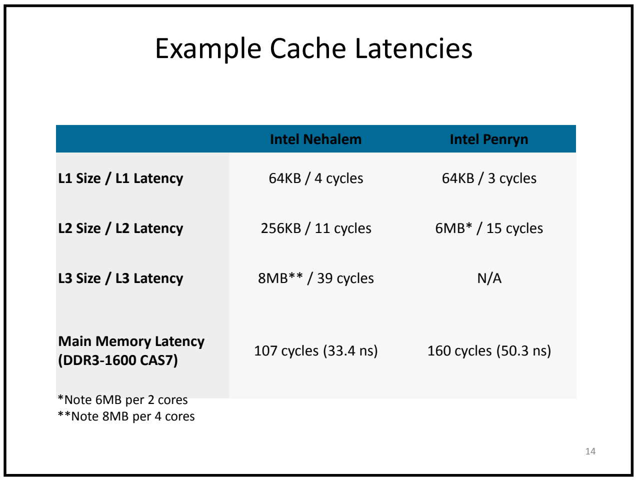
Cache Performance

- Let hit time = t_h, miss penalty = t_m, miss rate = m
- Suppose t_h = 1, t_m = 10
  - For m = .01, t_{acc} = 1.09
  - for m = .05, t_{acc} = 1.45
  - for m = .1, t_{acc} = 1.9
  - for m = .25, t_{acc} = 3.25
  - for m = .5, t_{acc} = 5.5
  - for m = .75, t_{acc} = 7.75
- Bottom line: for low miss rates, effective memory performance approaches that of the cache
- Key to cache memory design is to minimize the miss rate.



Memory Hierarchy

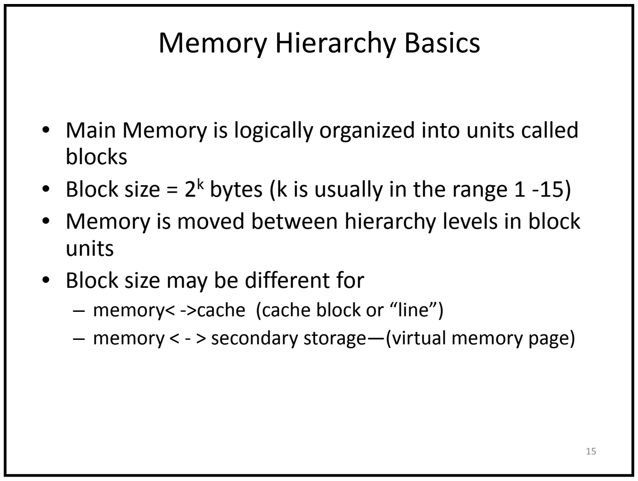
Temporal Locality
•Keep recently referenced items at higher levels
•Future references satisfied quickly
CPU
I & D L1 Cache
Shared L2 Cache
Main Memory
Disk
Spatial Locality
•Bring neighbors of recently referenced to higher levels
•Future references satisfied quickly



Example Cache Latencies

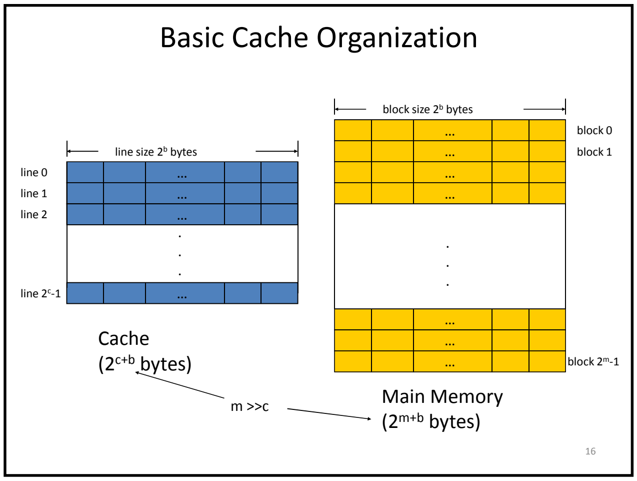
	Intel Nehalem	Intel Penryn
L1 Size / L1 Latency	64KB / 4 cycles	64KB / 3 cycles
L2 Size / L2 Latency	256KB / 11 cycles	6MB* / 15 cycles
L3 Size / L3 Latency	8MB** / 39 cycles	N/A
Main Memory Latency (DDR3-1600 CAS7)	107 cycles (33.4 ns)	160 cycles (50.3 ns)

*Note 6MB per 2 cores
**Note 8MB per 4 cores

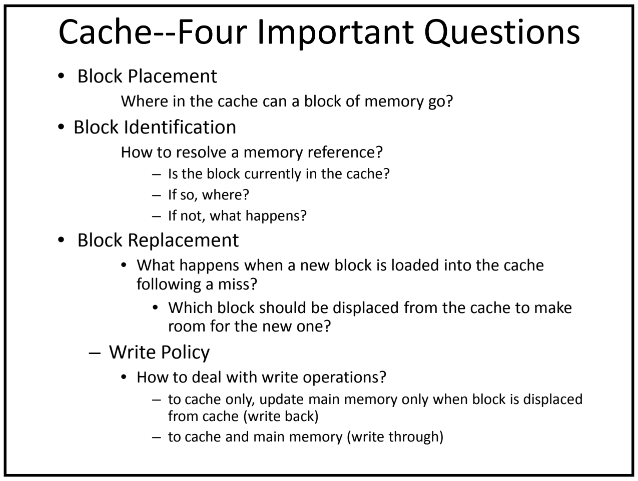


Memory Hierarchy Basics

- Main Memory is logically organized into units called blocks
- Block size = 2^k bytes (k is usually in the range 1 -15)
- Memory is moved between hierarchy levels in block units
- Block size may be different for
  - memory< ->cache (cache block or “line”)
  - memory < - > secondary storage—(virtual memory page)



Basic Cache Organization

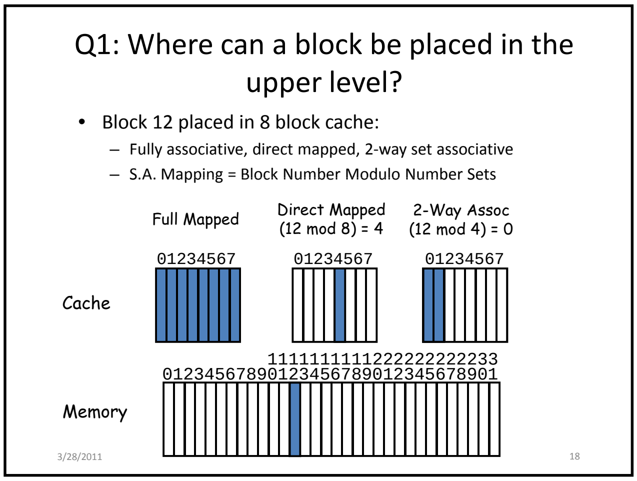


Cache--Four Important Questions

- Block Placement

  Where in the cache can a block of memory go?
- Block Identification

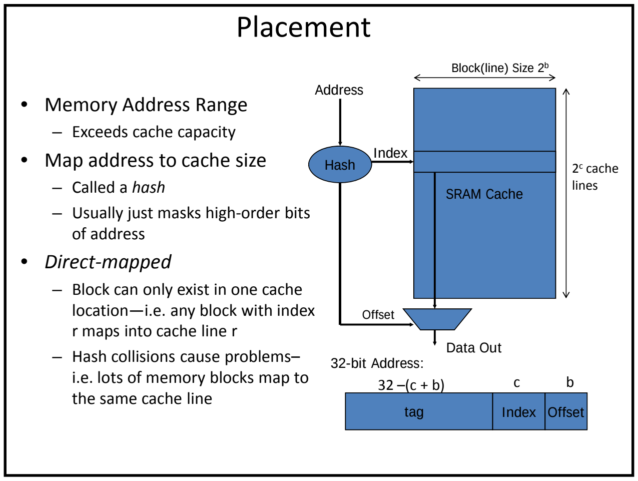
  How to resolve a memory reference?
  - Is the block currently in the cache?
  - If so, where?
  - If not, what happens?
- Block Replacement
  - What happens when a new block is loaded into the cache following a miss?
    - Which block should be displaced from the cache to make room for the new one?
  - Write Policy
    - How to deal with write operations?
      - to cache only, update main memory only when block is displaced from cache (write back)
      - to cache and main memory (write through)



Q1: Where can a block be placed in the upper level?

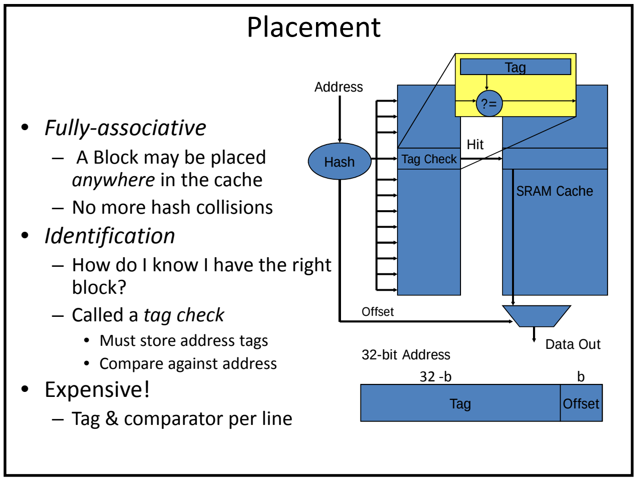
- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets

3/28/2011 18



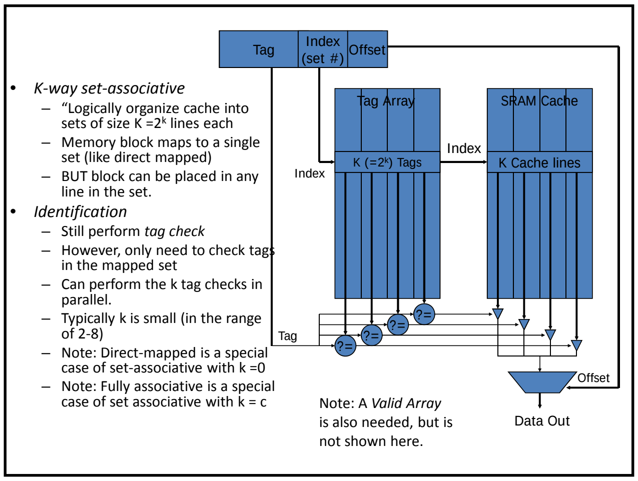
Placement

- Memory Address Range
  - Exceeds cache capacity
- Map address to cache size
  - Called a *hash*
  - Usually just masks high-order bits of address
- *Direct-mapped*
  - Block can only exist in one cache location—i.e. any block with index r maps into cache line r
  - Hash collisions cause problems–i.e. lots of memory blocks map to the same cache line



Placement

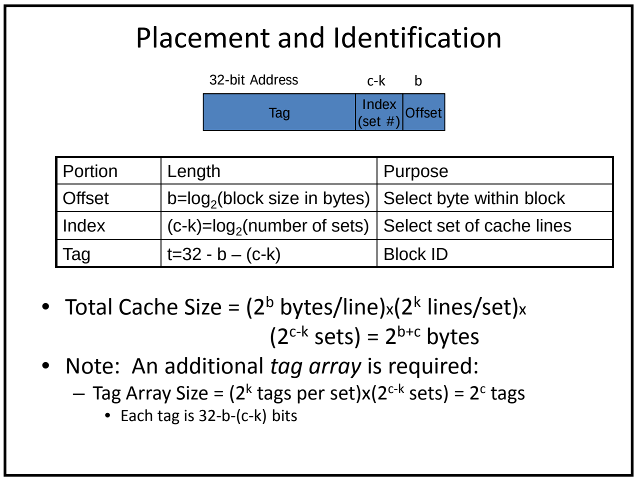
- *Fully-associative*
  - A Block may be placed *anywhere* in the cache
  - No more hash collisions
- *Identification*
  - How do I know I have the right block?
  - Called a *tag check*
    - Must store address tags
    - Compare against address
- Expensive!
  - Tag & comparator per line



- *K-way set-associative*
  - "Logically organize cache into sets of size K =2^k lines each
  - Memory block maps to a single set (like direct mapped)
  - BUT block can be placed in any line in the set.
- *Identification*
  - Still perform *tag check*
  - However, only need to check tags in the mapped set
  - Can perform the k tag checks in parallel.
  - Typically k is small (in the range of 2-8)
  - Note: Direct-mapped is a special case of set-associative with k =0
  - Note: Fully associative is a special case of set associative with k = c

Note: A *Valid Array* is also needed, but is not shown here.

Tag
Index (set #)
Offset
Tag Array
SRAM Cache
Index
K (=2^k) Tags
Index
K Cache lines
?=
?=
?=
?=
Tag
Offset
Data Out

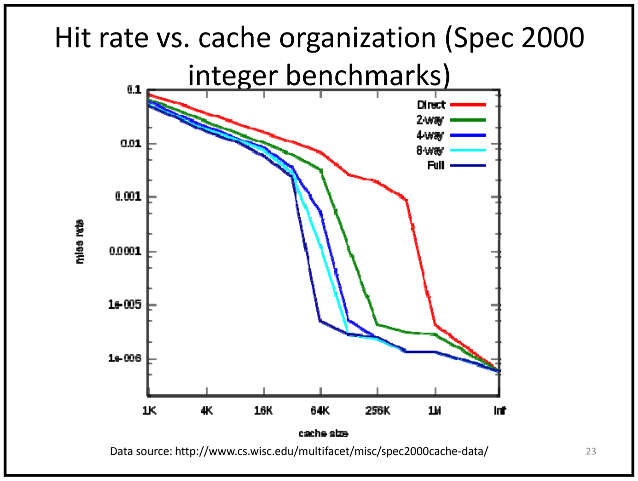


Placement and Identification

32-bit Address
c-k
b
Tag
Index (set #)
Offset

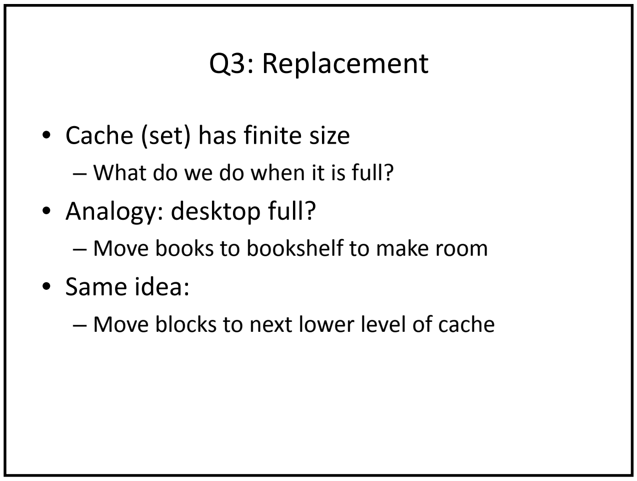
Portion	Length	Purpose
Offset	b=$\log_2$(block size in bytes)	Select byte within block
Index	(c-k)=$\log_2$(number of sets)	Select set of cache lines
Tag	t=32 - b – (c-k)	Block ID

- Total Cache Size = (2^b bytes/line)x(2^k lines/set)x (2^{c-k} sets) = 2^{b+c} bytes
- Note: An additional *tag array* is required:
  - Tag Array Size = (2^k tags per set)x(2^{c-k} sets) = 2^c tags
    - Each tag is 32-b-(c-k) bits



Hit rate vs. cache organization (Spec 2000 integer benchmarks)

Data source: http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/



Q3: Replacement

- Cache (set) has finite size
  - What do we do when it is full?
- Analogy: desktop full?
  - Move books to bookshelf to make room
- Same idea:
  - Move blocks to next lower level of cache

Replacement

- How do we choose *victim* to be replaced?
- Several policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used)
  - NMRU (not most recently used)
  - Pseudo-random (yes, really!)
- Pick victim within *set* where K = *associativity*
  - If K = 2, LRU is cheap and easy (1 bit)
  - If K > 2, it gets harder
  - Pseudo-random works pretty well for caches

Cache Replacement Policy Performance

- Easy for Direct Mapped (only one choice)
- Set Associative or Fully Associative:
  - Rand (Random)
  - LRU (Least Recently Used)

Assoc:	**2-way**		**4-way**		**8-way**	
Size	**LRU**	**Rand**	**LRU**	**Rand**	**LRU**	**Rand**
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%

Q4: Write Policy

- Memory hierarchy
  - 2 or more copies of same block
    - Cache/Main memory /disk
- What to do on a write?
  - Eventually, all copies must be changed
  - Write must *propagate* to all levels

Write Policies

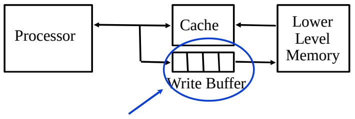
	Write-Through	Write-Back
Policy	Data written to cache block also written to lower-level memory	Write data only to the cache Update lower level when a block falls out of the cache
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Additional option -- let writes to an un-cached address allocate a new cache line (“write-allocate”).

Write Policy

- Easiest policy: *write-through*
- Every write propagates directly through hierarchy
  - Write in L1, L2, memory, disk (?!?)
- Drawbacks?
  - Very high bandwidth requirement
  - Remember, large memories are slow
- Popular in real systems only to the L2
  - Every write updates L1 and L2
  - Beyond L2, use *write-back* policy

Write Buffers for Write-Through Caches



Holds data awaiting write-through to lower level memory

Q. Why a write buffer ?

A. So CPU doesn't stall

Q. Why a buffer, why not just one register ?

A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?

A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

Write Policy

- Most widely used: *write-back*
- Maintain *state* of each line in a cache
  - Invalid – not present in the cache
  - Clean – present, but not written (unmodified)
  - Dirty – present and written (modified)
- Store state in tag array, next to address tag
  - Mark dirty bit on a write
- On eviction, check dirty bit
  - If set, write back dirty line to next level
  - Called a *write-back* or *cast-out*

Write Policy

- Complications of write-back policy
  - Stale copies lower in the hierarchy
  - Must always check higher level for dirty copies before accessing copy in a lower level
- Not a big problem in uniprocessors
  - In multiprocessors: *the cache coherence problem*
- I/O devices that use DMA (direct memory access) can cause problems even in uniprocessors
  - Called coherent I/O
  - Must check caches for dirty copies before reading main memory

Caches and Performance

- Caches
  - Enable design for common case: cache hit
    - Cycle time, pipeline organization
    - Recovery policy
  - Uncommon case: cache miss
    - Fetch from next level
      - Apply recursively if multiple levels
    - What to do in the meantime?
- What is performance impact?
- Various optimizations are possible

Cache Performance Impact

- Cache hit latency
  - Included in “pipeline” portion of CPI
  - Typically 1-3 cycles for L1 cache
    - Intel/HP McKinley: 1 cycle
      - Heroic array design
      - No address generation: load r1, (r2)
    - IBM Power4: 3 cycles
      - Address generation
      - Array access
      - Word select and align

Example Cache Latencies

L1 cache hit latency

	Intel Nehalem	Intel Penryn
L1 Size / L1 Latency	64KB / 4 cycles	64KB / 3 cycles
L2 Size / L2 Latency	256KB / 11 cycles	6MB* / 15 cycles
L3 Size / L3 Latency	8MB** / 39 cycles	N/A
Main Memory Latency (DDR3-1600 CAS7)	107 cycles (33.4 ns)	160 cycles (50.3 ns)

*Note 6MB per 2 cores
**Note 8MB per 4 cores

Cache Misses and Performance

- Miss penalty
  - Detect miss: 1 or more cycles
  - Find victim (replace line): 1 or more cycles
    - Write back if dirty
  - Request line from next level: several cycles
  - Transfer line from next level: several cycles
    - (block size) / (bus width)
  - Fill line into data array, update tag array: 1+ cycles
  - Resume execution
- In practice: 6 cycles to 100s of cycles

Example Cache Latencies

Miss penalty for L1 miss

	Intel Nehalem	Intel Penryn
L1 Size / L1 Latency	64KB / 4 cycles	64KB / 3 cycles
L2 Size / L2 Latency	256KB / 11 cycles	6MB* / 15 cycles
L3 Size / L3 Latency	8MB** / 39 cycles	N/A
Main Memory Latency (DDR3-1600 CAS7)	107 cycles (33.4 ns)	160 cycles (50.3 ns)

*Note 6MB per 2 cores
**Note 8MB per 4 cores

37

Cache Miss Rate

- Determined by:
  - Program characteristics
    - Temporal locality
    - Spatial locality
  - Cache organization
    - Block size, associativity, number of sets

Improving Locality

- Instruction text placement
  - Profile program, place unreferenced or rarely referenced paths “elsewhere”
    - Maximize temporal locality
  - Eliminate taken branches
    - Fall-through path has spatial locality

Improving Locality

- Data placement, access order
  - Arrays: “block” loops to access subarray that fits into cache
    - Maximize temporal locality
  - Structures: pack commonly-accessed fields together
    - Maximize spatial, temporal locality
  - Trees, linked lists: allocate in usual reference order
    - Heap manager usually allocates sequential addresses
    - Maximize spatial locality
- Hard problem, not easy to automate:
  - C/C++ disallows rearranging structure fields
  - OK in Java

Cache Miss Rates: 3 C's [Hill]

- Compulsory miss
  - First-ever reference to a given block of memory
- Capacity
  - Working set exceeds cache capacity
  - Useful blocks (with future references) displaced
- Conflict
  - Placement restrictions (not fully-associative) cause useful blocks to be displaced
  - Think of as *capacity within set*

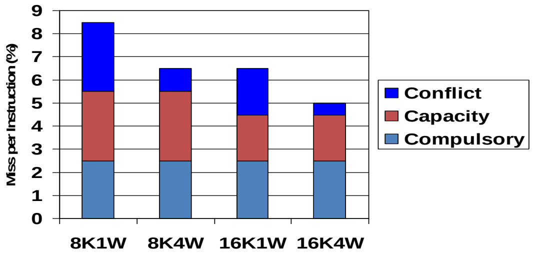
Cache Miss Rate Effects

- Number of blocks (sets x associativity)
  - Bigger is better: fewer conflicts, greater capacity
- Associativity
  - Higher associativity reduces conflicts
  - Very little benefit beyond 8-way set-associative
- Block size
  - Larger blocks exploit spatial locality
  - Usually: miss rates improve until 64B-256B
  - 512B or more miss rates get worse
    - Larger blocks less efficient: more capacity misses
    - Fewer placement choices: more conflict misses

Cache Miss Rate

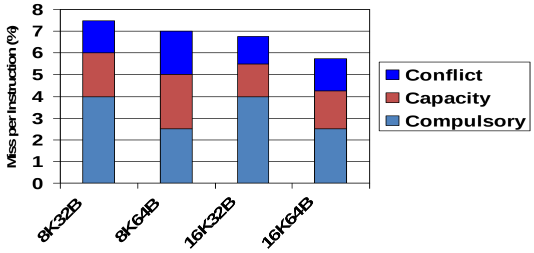
- Subtle tradeoffs between cache organization parameters
  - Large blocks reduce compulsory misses but increase miss penalty
    - #compulsory = (working set) / (block size)
    - #transfers = (block size)/(bus width)
  - Large blocks increase conflict misses
    - #blocks = (cache size) / (block size)
  - Associativity reduces conflict misses
  - Associativity increases access time
- Can associative cache ever have higher miss rate than direct-mapped cache of same size?

Cache Miss Rates: 3 C's



- Vary size and associativity
  - Compulsory misses are constant
  - Capacity and conflict misses are reduced

Cache Miss Rates: 3 C's



- Vary size and block size
  - Compulsory misses drop with increased block size
  - Capacity and conflict can increase with larger blocks

Cache Misses and Performance

- How does this affect performance?
- Performance = Time / Program

$$= \frac{\textbf{Instructions}}{\textbf{Program}} \;\textbf{X}\; \frac{\textbf{Cycles}}{\textbf{Instruction}} \;\textbf{X}\; \frac{\textbf{Time}}{\textbf{Cycle}}$$

(code size) **(CPI)** **(cycle time)**

- Cache organization affects cycle time
  - Hit latency
- Cache misses affect **CPI**

Cache Misses and CPI

$$\begin{aligned} CPI &= \frac{cycles}{inst} = \frac{cycles_{hit}}{inst} + \frac{cycles_{miss}}{inst} \\ &= \frac{cycles_{hit}}{inst} + \frac{cycles}{miss} \times \frac{miss}{inst} \\ &= \frac{cycles_{hit}}{inst} + Miss\_penalty \times Miss\_rate \end{aligned}$$

- Cycles spent handling misses are strictly additive
- Miss_penalty is recursively defined at next level of cache hierarchy as weighted sum of hit latency and miss latency

Cache Misses and CPI

$$CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l$$

- P_l is miss penalty at each of n levels of cache
- MPI_l is miss rate per instruction at each of n levels of cache
- Miss rate specification:
  - Per instruction: easy to incorporate in CPI
  - Per reference: must convert to per instruction
    - Local: misses per local reference
    - Global: misses per ifetch or load or store

Cache Performance Example

- Assume following:
  - L1 instruction cache with 98% per instruction hit rate
  - L1 data cache with 96% per instruction hit rate
  - Shared L2 cache with 40% local miss rate
  - L1 miss penalty of 8 cycles
  - L2 miss penalty of:
    - 10 cycles latency to request word from memory
    - 2 cycles per 16B bus transfer, 4x16B = 64B block transferred
    - Hence 8 cycles transfer plus 1 cycle to fill L2
    - Total penalty 10+8+1 = 19 cycles

Cache Performance Example

$$CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l$$

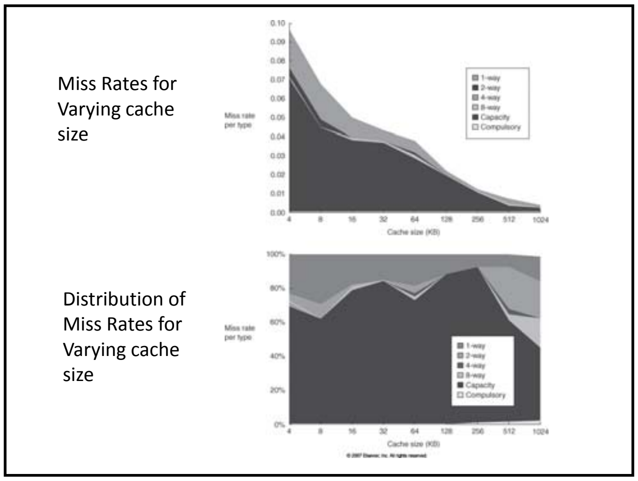
$$\begin{aligned} CPI &= 1.15 + \frac{8cycles}{miss} \times \left(\frac{0.02miss}{inst} + \frac{0.04miss}{inst} \right) \\ &+ \frac{19cycles}{miss} \times \frac{0.40miss}{ref} \times \frac{0.06ref}{inst} \\ &= 1.15 + 0.48 + \frac{19cycles}{miss} \times \frac{0.024miss}{inst} \\ &= 1.15 + 0.48 + 0.456 = 2.086 \end{aligned}$$

Cache Misses and Performance

- CPI equation
  - Only holds for misses that cannot be overlapped with other activity
  - Store misses often overlapped
    - Place store in store queue
    - Wait for miss to complete
    - Perform store
    - Allow subsequent instructions to continue in parallel
  - Modern out-of-order processors also do this for loads
    - Cache performance modeling requires detailed modeling of entire processor core

5 Basic Cache Optimizations

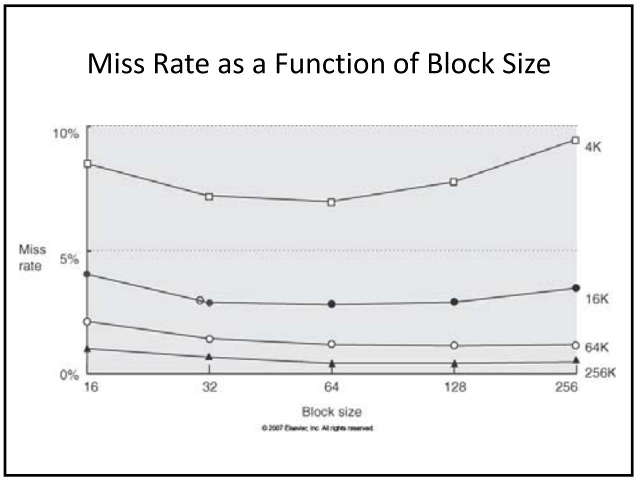
- Reducing Miss Rate
  1. Larger Block size (compulsory misses)
  2. Larger Cache size (capacity misses)
  3. Higher Set Associativity (conflict misses)
- Reducing Miss Penalty
  4. Multilevel Caches
- Reducing hit time
  5. Giving Reads Priority over Writes
     - E.g., Read complete before earlier writes in write buffer



Miss Rates for Varying cache size

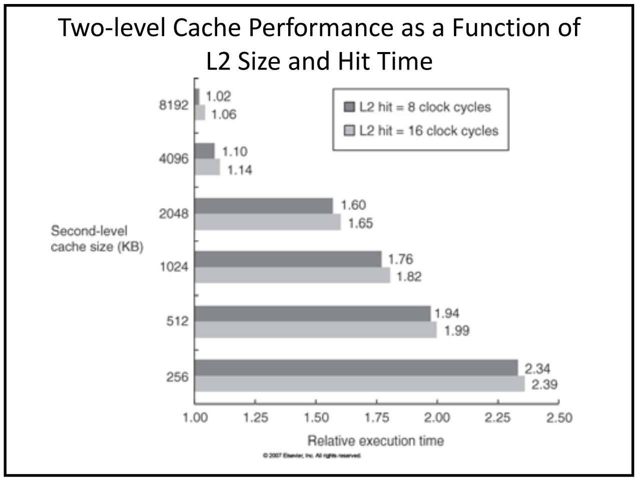
Distribution of Miss Rates for Varying cache size

© 2007 Elsevier, Inc. All rights reserved.



Miss Rate as a Function of Block Size

© 2007 Elsevier, Inc. All rights reserved.



Two-level Cache Performance as a Function of L2 Size and Hit Time

© 2007 Elsevier, Inc. All rights reserved.

Main Memory

- Memory organization
  - Interleaving
  - Banking
- Memory controller design

Simple Main Memory

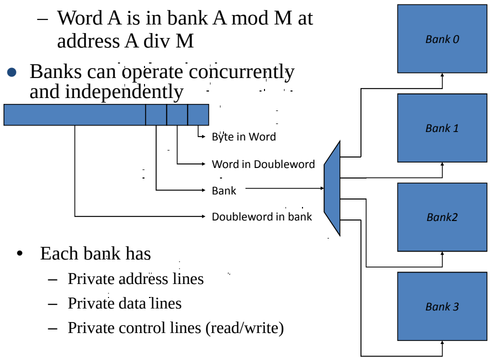
- Consider these parameters:
  - 1 cycle to send address
  - 6 cycles to access each word
  - 1 cycle to send word back
- Miss penalty for a 4-word block
  - (1 + 6 + 1) x 4 = 32
- How can we speed this up?

Wider(Parallel) Main Memory

- Make memory wider
  - Read out all words in parallel
- Memory parameters
  - 1 cycle to send address
  - 6 to access a double word
  - 1 cycle to send it back
- Miss penalty for 4-word block: 1+6+1 = 16
- Costs
  - Wider bus
  - Larger minimum expansion unit

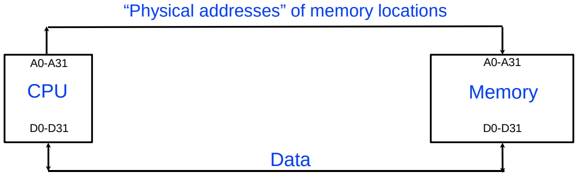
Interleaved Main Memory

- Break memory into M banks
  - Word A is in bank A mod M at address A div M
- Banks can operate concurrently and independently



- Each bank has
  - Private address lines
  - Private data lines
  - Private control lines (read/write)

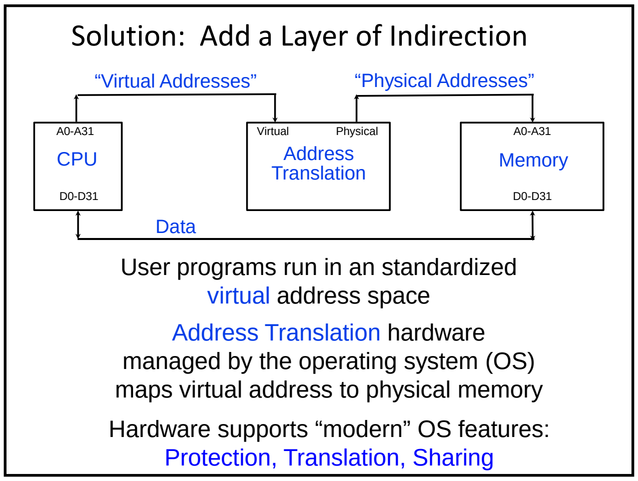
The Limits of Physical Addressing



All programs share one address space:
The physical address space

Machine language programs must be aware of the machine organization

No way to prevent a program from accessing any machine resource

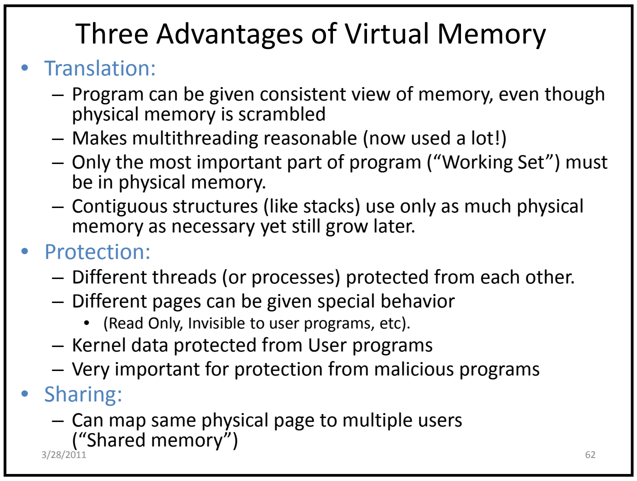


Solution: Add a Layer of Indirection

User programs run in an standardized virtual address space

Address Translation hardware managed by the operating system (OS) maps virtual address to physical memory

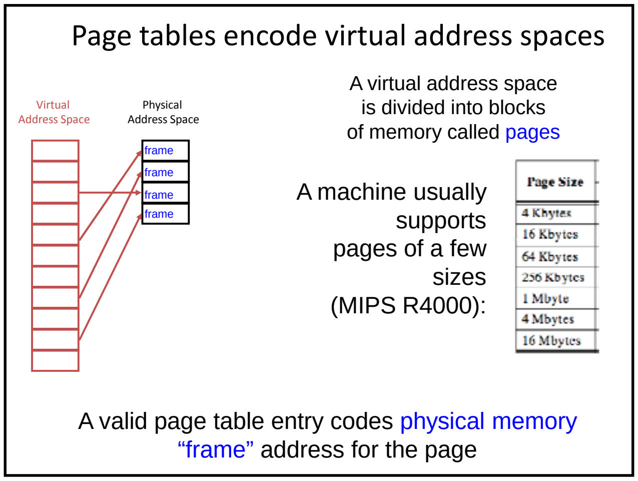
Hardware supports "modern" OS features: Protection, Translation, Sharing



Three Advantages of Virtual Memory

- Translation:
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Makes multithreading reasonable (now used a lot!)
  - Only the most important part of program ("Working Set") must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.
- Protection:
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior
    - (Read Only, Invisible to user programs, etc).
  - Kernel data protected from User programs
  - Very important for protection from malicious programs
- Sharing:
  - Can map same physical page to multiple users ("Shared memory")

3/28/2011 62



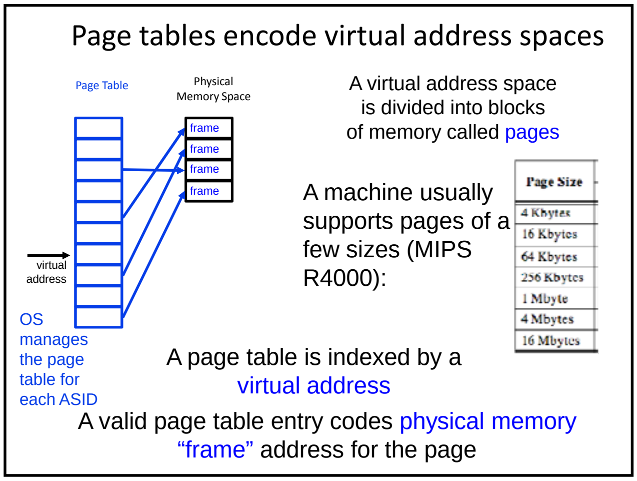
Page tables encode virtual address spaces

A virtual address space is divided into blocks of memory called pages

A machine usually supports pages of a few sizes (MIPS R4000):

Page Size
4 Kbytes
16 Kbytes
64 Kbytes
256 Kbytes
1 Mbyte
4 Mbytes
16 Mbytes

A valid page table entry codes physical memory "frame" address for the page



Page tables encode virtual address spaces

OS manages the page table for each ASID

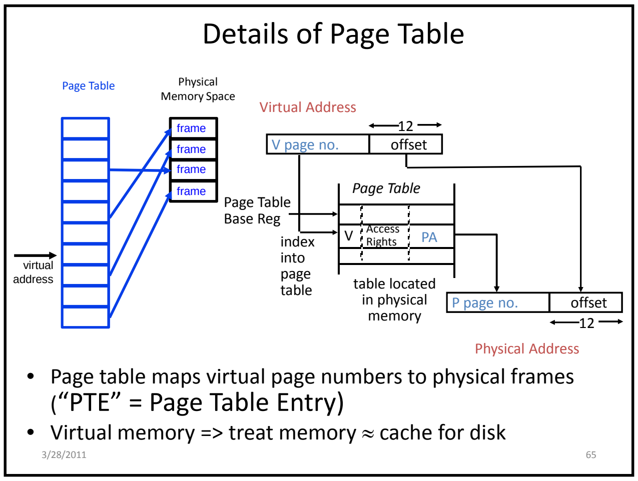
A virtual address space is divided into blocks of memory called pages

A machine usually supports pages of a few sizes (MIPS R4000):

Page Size
4 Kbytes
16 Kbytes
64 Kbytes
256 Kbytes
1 Mbyte
4 Mbytes
16 Mbytes

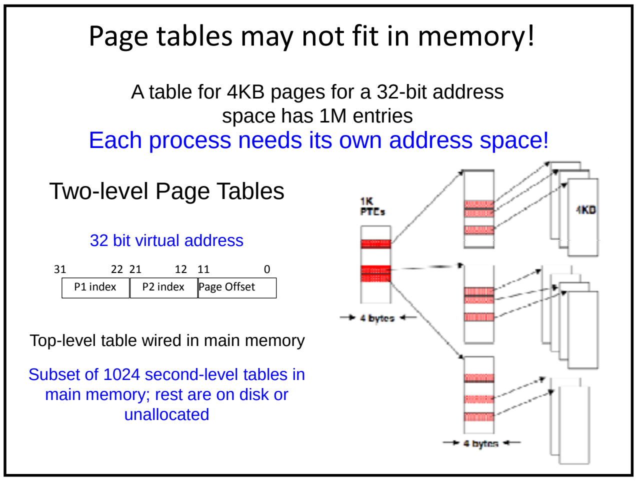
A page table is indexed by a virtual address

A valid page table entry codes physical memory "frame" address for the page



Details of Page Table

- Page table maps virtual page numbers to physical frames (“PTE” = Page Table Entry)
- Virtual memory => treat memory ≈ cache for disk



Page tables may not fit in memory!

A table for 4KB pages for a 32-bit address space has 1M entries

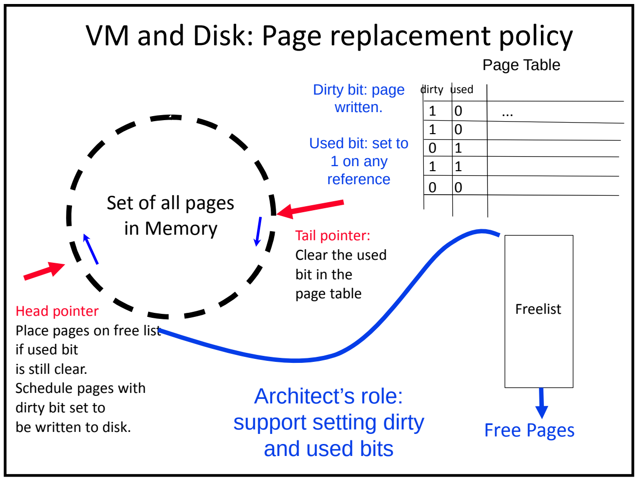
Each process needs its own address space!

Two-level Page Tables

32 bit virtual address

Top-level table wired in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated



VM and Disk: Page replacement policy

Page Table

dirty	used	
1	0	...
1	0	
0	1	
1	1	
0	0	

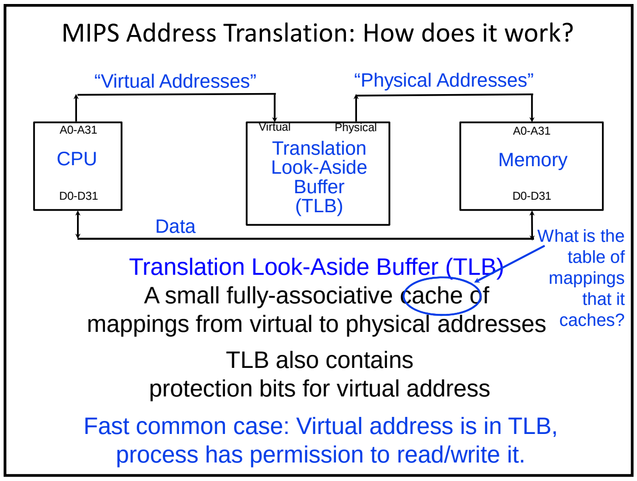
Dirty bit: page written.

Used bit: set to 1 on any reference

Tail pointer: Clear the used bit in the page table

Head pointer Place pages on free list if used bit is still clear. Schedule pages with dirty bit set to be written to disk.

Architect’s role: support setting dirty and used bits



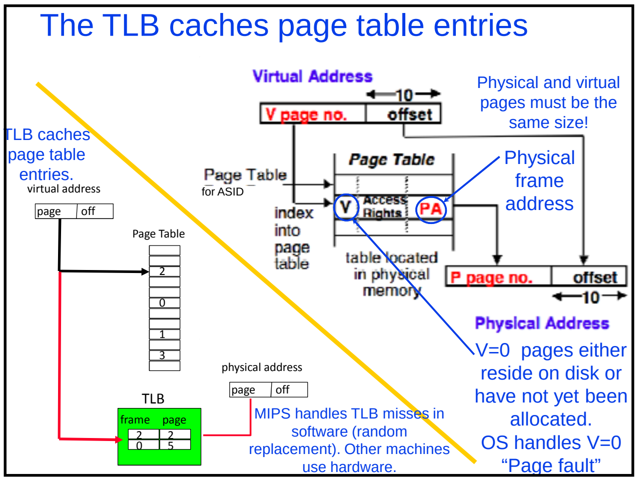
MIPS Address Translation: How does it work?

What is the table of mappings that it caches?

Translation Look-Aside Buffer (TLB) A small fully-associative cache of mappings from virtual to physical addresses

TLB also contains protection bits for virtual address

Fast common case: Virtual address is in TLB, process has permission to read/write it.



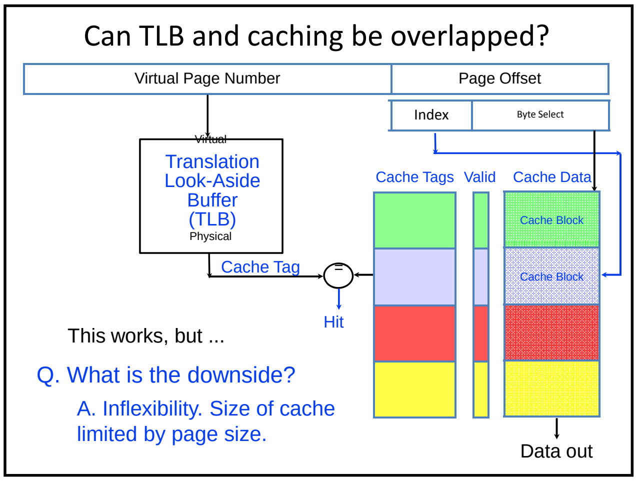
The TLB caches page table entries

TLB caches page table entries.

Physical and virtual pages must be the same size!

MIPS handles TLB misses in software (random replacement). Other machines use hardware.

V=0 pages either reside on disk or have not yet been allocated.
OS handles V=0
“Page fault”

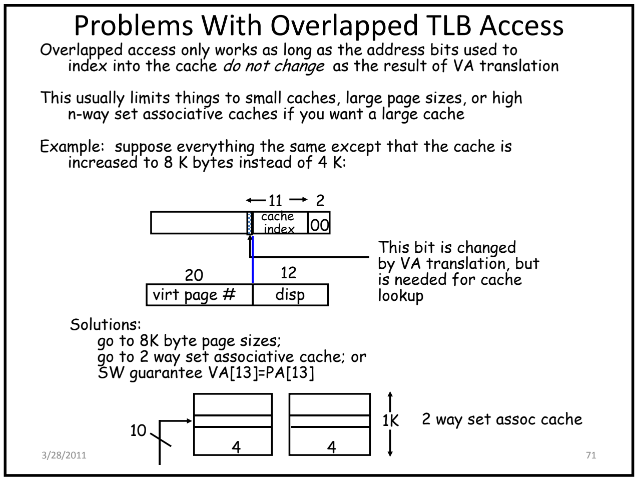


Can TLB and caching be overlapped?

This works, but ...

Q. What is the downside?

A. Inflexibility. Size of cache limited by page size.



Problems With Overlapped TLB Access

Overlapped access only works as long as the address bits used to index into the cache *do not change* as the result of VA translation

This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache

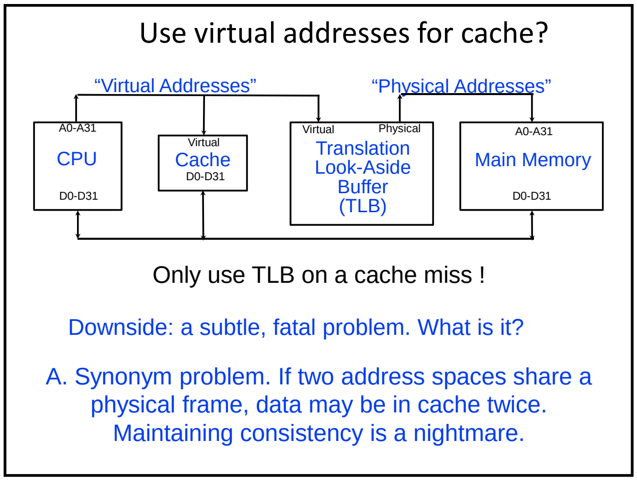
Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:

This bit is changed by VA translation, but is needed for cache lookup

Solutions:
go to 8K byte page sizes;
go to 2 way set associative cache; or
SW guarantee VA[13]=PA[13]

2 way set assoc cache

3/28/2011 71



Use virtual addresses for cache?

Only use TLB on a cache miss !

Downside: a subtle, fatal problem. What is it?

A. Synonym problem. If two address spaces share a physical frame, data may be in cache twice. Maintaining consistency is a nightmare.

Advanced Cache Optimizations

- Reducing hit time

1. Small and simple caches
2. Way prediction
3. Trace caches

- Increasing cache bandwidth

4. Pipelined caches
5. Multibanked caches
6. Nonblocking caches

- Reducing Miss Penalty

7. Critical word first
8. Merging write buffers

- Reducing Miss Rate

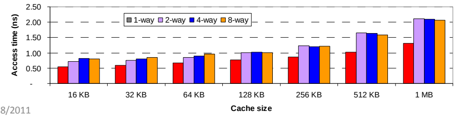
9. Compiler optimizations

- Reducing miss penalty or miss rate via parallelism

10. Hardware prefetching
11. Compiler prefetching

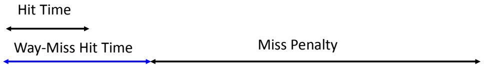
1. Fast Hit times via Small and Simple Caches

- Index tag memory and then compare takes time
- ⇒ Small cache can help hit time since smaller memory takes less time to index
  - E.g., L1 caches same size for 3 generations of AMD microprocessors: K6, Athlon, and Opteron
  - Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip
- Simple ⇒ direct mapping
  - Can overlap tag check with data transmission since no choice
- Access time estimate for 90 nm using CACTI model 4.0
  - Median ratios of access time relative to the direct-mapped caches are 1.32, 1.39, and 1.43 for 2-way, 4-way, and 8-way caches



2. Fast Hit times via Way Prediction

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Way prediction: keep extra bits in cache to predict the "way," or block within the set, of next cache access.
  - Multiplexor is set early to select desired block, only 1 tag comparison performed that clock cycle in parallel with reading the cache data
  - Miss ⇒ 1st check other blocks for matches in next clock cycle



- Accuracy ≈ 85%
- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  - Used for instruction caches vs. data caches

3. Fast Hit times via Trace Cache (Pentium 4 only; and last time?)

- Find more instruction level parallelism?
  How avoid translation from x86 to microops?
- Trace cache in Pentium 4

1. Dynamic traces of the executed instructions vs. static sequences of instructions as determined by layout in memory
   - Built-in branch predictor
2. Cache the micro-ops vs. x86 instructions
   - Decode/translate from x86 to micro-ops on trace cache miss

+ 1. ⇒ better utilize long blocks (don't exit in middle of block, don't enter at label in middle of block)

- 1. ⇒ complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size

- 1. ⇒ instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

4: Increasing Cache Bandwidth by Pipelining

- Pipeline cache access to maintain bandwidth, but higher latency
- Instruction cache access pipeline stages:

  1: Pentium

  2: Pentium Pro through Pentium III

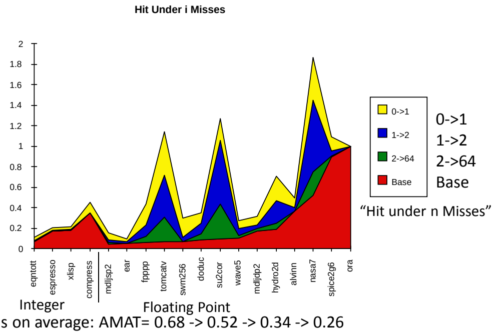
  4: Pentium 4

- ⇒ greater penalty on mispredicted branches
- ⇒ more clock cycles between the issue of the load and the use of the data

5. Increasing Cache Bandwidth: Non-Blocking Caches

- *Non-blocking cache* or *lockup-free cache* allow data cache to continue to supply cache hits during a miss
  - requires F/E bits on registers or out-of-order execution
  - requires multi-bank memories
- "*hit under miss*" reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  - Requires muliple memory banks (otherwise cannot support)
  - Penium Pro allows 4 outstanding memory misses

Value of Hit Under Miss for SPEC (old data)



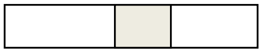
- FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26
- Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19
- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss, SPEC 92

6: Increasing Cache Bandwidth via Multiple Banks

- Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses
  - E.g.,T1 ("Niagara") L2 has 4 banks
- Banking works best when accesses naturally spread themselves across banks ⇒ mapping of addresses to banks affects behavior of memory system
- Simple mapping that works well is "sequential interleaving"
  - Spread block addresses sequentially across banks
  - E,g, if there 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; …

7. Reduce Miss Penalty: Early Restart and Critical Word First

- Don’t wait for full block before restarting CPU
- *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Spatial locality ⇒ tend to want next sequential word, so not clear size of benefit of just early restart
- *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 the rest of the words in the block
  - Long blocks more popular today ⇒ Critical Word 1st Widely used



block

8. Merging Write Buffer to Reduce Miss Penalty

- Write buffer to allow processor to continue while waiting to write to memory
- If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry
- If so, new data are combined with that entry
- Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory
- The Sun T1 (Niagara) processor, among many others, uses write merging

9. Reducing Misses by Compiler Optimizations

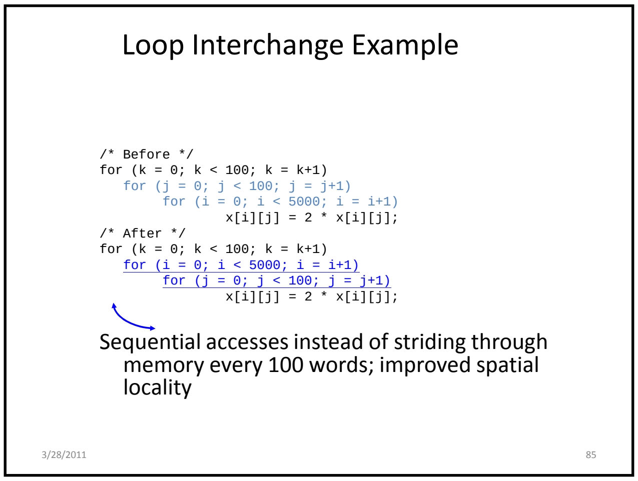
- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts(using tools they developed)
- Data
  - *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
   int val;
   int key;
};
struct merge merged_array[SIZE];
```

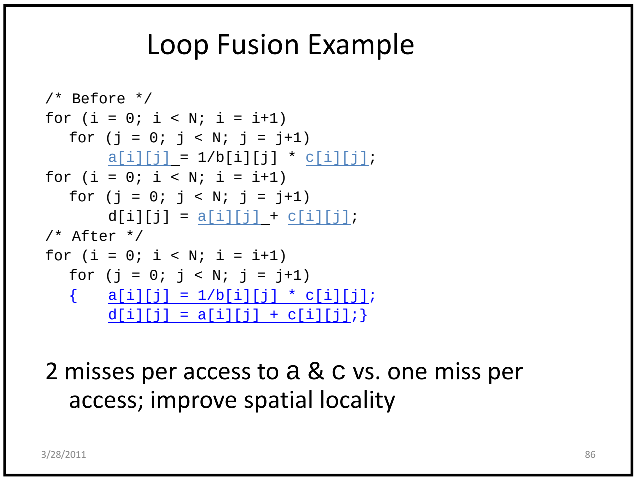
Reducing conflicts between val & key;
improve spatial locality



Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
   for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i = i+1)
                x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
   for (i = 0; i < 5000; i = i+1)
        for (j = 0; j < 100; j = j+1)
                x[i][j] = 2 * x[i][j];
```

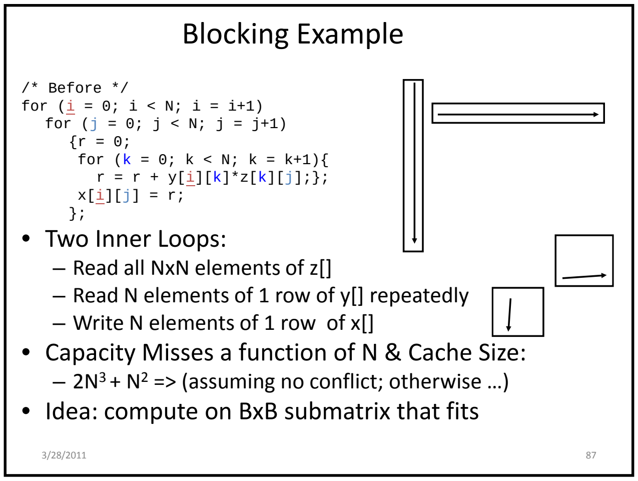
Sequential accesses instead of striding through memory every 100 words; improved spatial locality



Loop Fusion Example

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

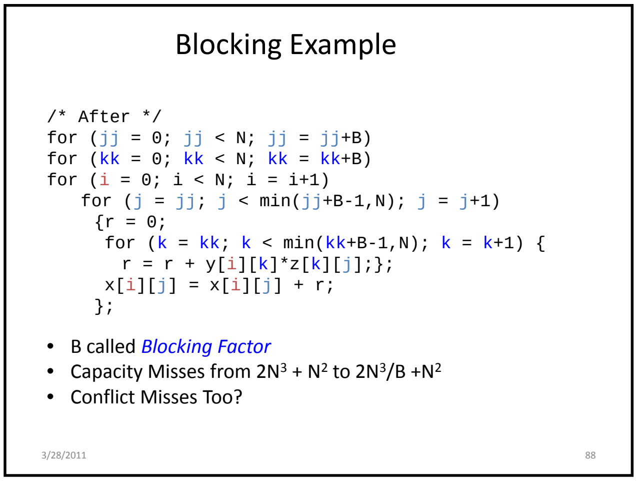
2 misses per access to a & c vs. one miss per access; improve spatial locality



Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
      {r = 0;
       for (k = 0; k < N; k = k+1){
         r = r + y[i][k]*z[k][j];};
       x[i][j] = r;
      };
```

- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]
- Capacity Misses a function of N & Cache Size:
  - $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

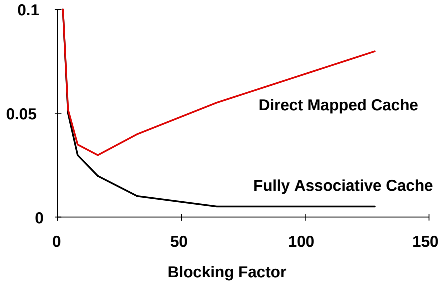


Blocking Example

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

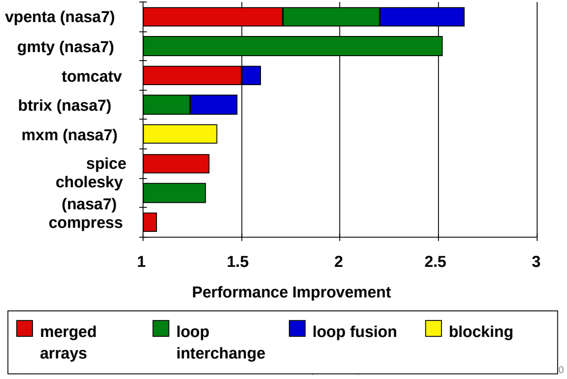
- B called *Blocking Factor*
- Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
- Conflict Misses Too?

Reducing Conflict Misses by Blocking



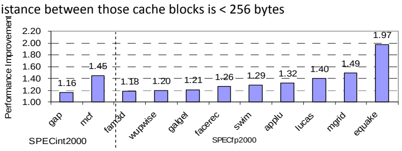
- Conflict misses in caches not FA vs. Blocking size
  - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache

Summary of Compiler Optimizations to Reduce Cache Misses (by hand)



10. Reducing Misses by Hardware Prefetching of Instructions & Data

- Prefetching relies on having extra memory bandwidth that can be used without penalty
- Instruction Prefetching
  - Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block.
  - Requested block is placed in instruction cache when it returns, and prefetched block is placed into instruction stream buffer
- Data Prefetching
  - Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages
  - Prefetching invoked if 2 successive L2 cache misses to a page, if distance between those cache blocks is < 256 bytes



11. Reducing Misses by Software Prefetching Data

- Data Prefetch
  - Load data into register (HP PA-RISC loads)
  - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
  - Special prefetching instructions cannot cause faults; a form of speculative execution
- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth