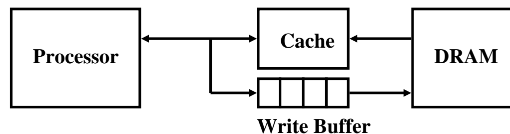
Traditional Four Questions for Memory Hierarchy Designers

- Q1: Where can a block be placed in the upper level? *(Block placement)*
  - Fully Associative, Set Associative, Direct Mapped
- Q2: How is a block found if it is in the upper level? *(Block identification)*
  - Tag/Block
- Q3: Which block should be replaced on a miss? *(Block replacement)*
  - Random, LRU
- Q4: What happens on a write? *(Write strategy)*
  - Write Back or Write Through (with Write Buffer)

Q4: What happens on a write?

- ***Write through***—The information is written to both the block in the cache and to the block in the lower-level memory.
- ***Write back***—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no repeated writes to same location
- WT always combined with write buffers so that don't wait for lower level memory

Write Buffer for Write Through



- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer's nightmare:
  - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
  - Write buffer saturation

What are all the aspects of cache organization that impact performance?

Review: Cache performance

- Miss-oriented Approach to Memory Access:

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right) \times CycleTime$$

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty \right) \times CycleTime$$

  - $CPI_{Execution}$ includes ALU and Memory instructions

- Separating out Memory component entirely
  - AMAT = Average Memory Access Time
  - CPI_{ALUOps} does not include memory instructions

$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

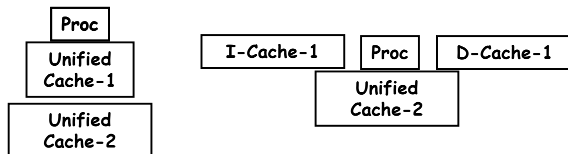
$$AMAT = HitTime + MissRate \times MissPenalty$$
$$= (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) +$$
$$(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data})$$

Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 10% of memory operations get 50 cycle miss penalty
- Suppose that 1% of instructions get same miss penalty
- CPI = ideal CPI + average stalls per instruction
  1.1(cycles/ins) +
  [0.30 (DataMops/ins)
  x 0.10 (miss/DataMop) x 50 (cycle/miss)] +
  [1 (InstMop/ins)
  x 0.01 (miss/InstMop) x 50 (cycle/miss)]
  = (1.1 + 1.5 + .5) cycle/ins = 3.1
- 58% of the time the proc is stalled waiting for memory!
- AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54

Unified vs Split Caches

- Unified vs Separate I&D



- Example:
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%
- Which is better (ignore L2 cache)?
  - Assume 33% data ops ⇒ 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that *data* hit has 1 stall for unified cache (only one port)

$AMAT_{Harvard}$=75%x(1+0.64%x50)+25%x(1+6.47%x50) = 2.05
$AMAT_{Unified}$=75%x(1+1.99%x50)+25%x(1+1+1.99%x50)= 2.24

How to Improve Cache Performance?

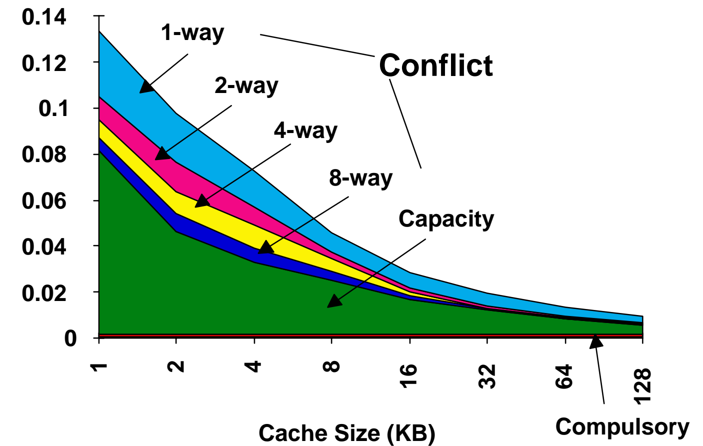
$$AMAT = HitTime + MissRate \times MissPenalty$$

1. ***Reduce the miss rate,***
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

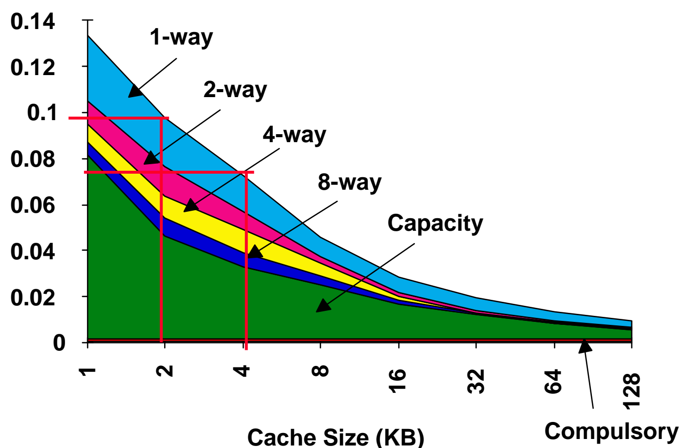
Where do misses come from?

- Classifying Misses: 3 Cs
  - *Compulsory*—The first access to a block is not in the cache, so the block must be brought into the cache. Also called *cold start misses* or *first reference misses*.
    (Misses in even an Infinite Cache)
  - *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.
    (Misses in Fully Associative Size X Cache)
  - *Conflict*—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses*.
    (Misses in N-way Associative, Size X Cache)
- 4th "C":
  - *Coherence* - Misses caused by cache coherence.

3Cs Absolute Miss Rate (SPEC92)



Cache Size



- Old rule of thumb: 2x size => 25% cut in miss rate
- What does it reduce?

Cache Organization?

- Assume total cache size not changed:
- What happens if:

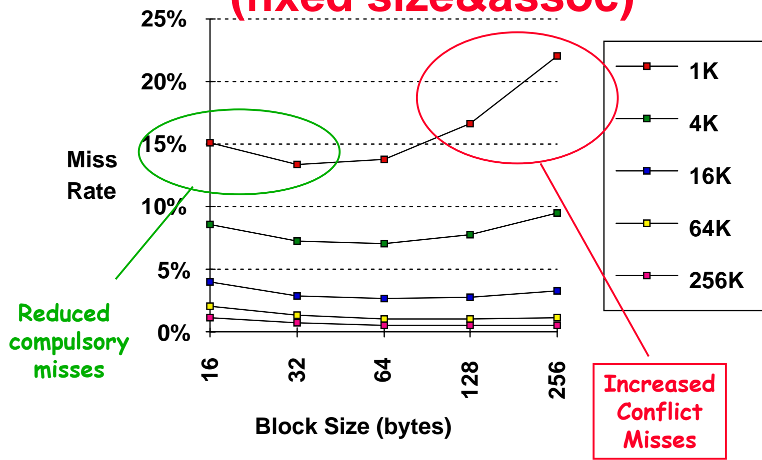
1) Change Block Size:

2) Change Associativity:

3) Change Compiler:

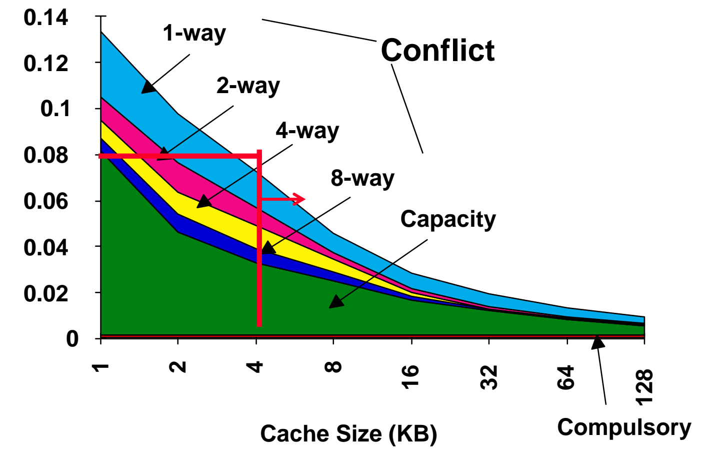
Which of 3Cs is obviously affected?

Larger Block Size (fixed size&assoc)

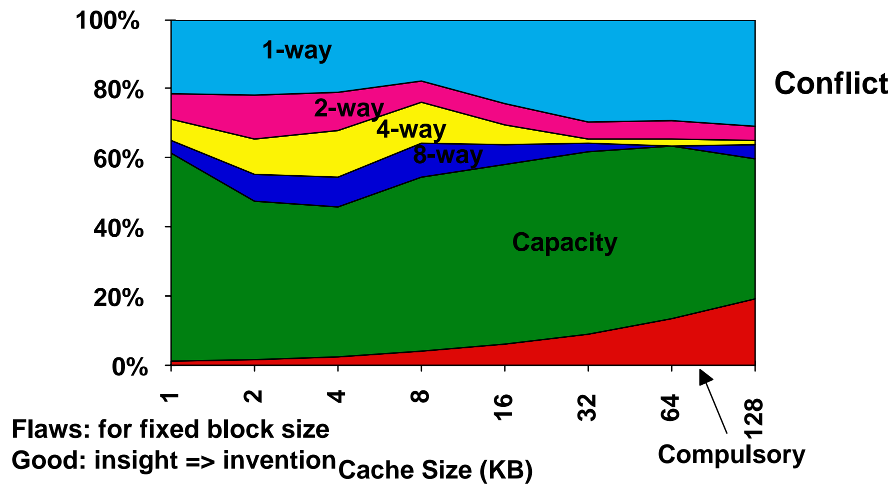


What else drives up block size?

Associativity



3Cs Relative Miss Rate



Flaws: for fixed block size
Good: insight => invention

Associativity vs Cycle Time

- Beware: Execution time is only final measure!
- Why is cycle time tied to hit time?
- Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%
  - suggested big and dumb caches

Effective cycle time of assoc
pzrbski ISCA

Example: Avg. Memory Access Time vs. Miss Rate

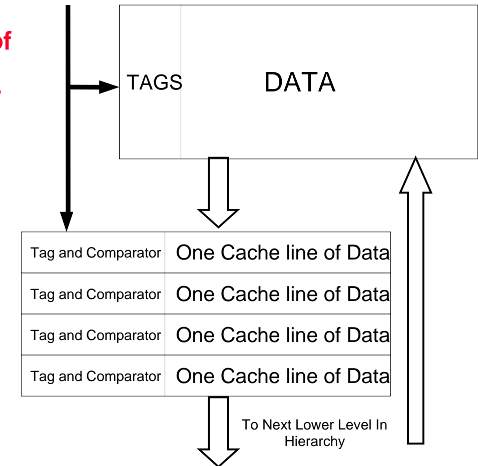
- Example: assume CCT = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. CCT direct mapped

Cache Size	Associativity			
(KB)	1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

(Red means A.M.A.T. not improved by more associativity)

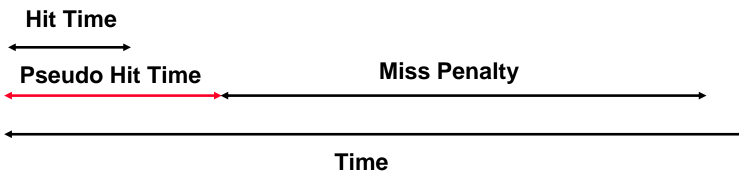
Fast Hit Time + Low Conflict => Victim Cache

- How to combine fast hit time of direct mapped yet still avoid conflict misses?
- Add buffer to place data discarded from cache
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
- Used in Alpha, HP machines



Reducing Misses via "Pseudo-Associativity"

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)



- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  - Better for caches not tied directly to processor (L2)
  - Used in MIPS R1000 L2 cache, similar in UltraSPARC

Reducing Misses by Hardware Prefetching of Instructions & Data

- E.g., Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in "stream buffer"
  - On miss check stream buffer
- Works with data blocks too:
  - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
  - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches
- Prefetching relies on having extra memory bandwidth that can be used without penalty

Reducing Misses by <u>Software</u> 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
- Prefetching comes in two flavors:
  - Binding prefetch: Requests load directly into register.
    - » Must be correct address and register!
  - Non-Binding prefetch: Load into cache.
    - » Can be incorrect. Faults?
- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth

Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks <u>in software</u>
- 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

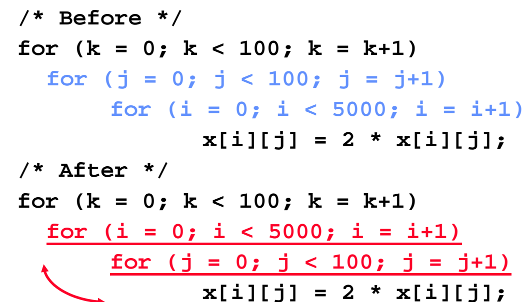
Merging Arrays Example

```
/* 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



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

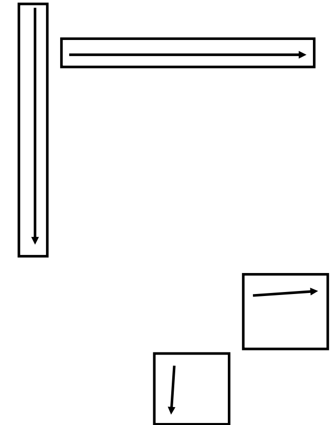
Loop Fusion Example

```
/* 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;
    };
```



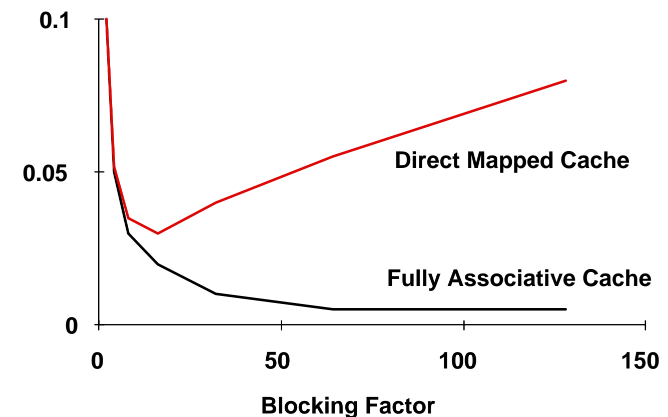
- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]
- Capacity Misses a function of N & Cache Size:
  - $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

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;
     };
```

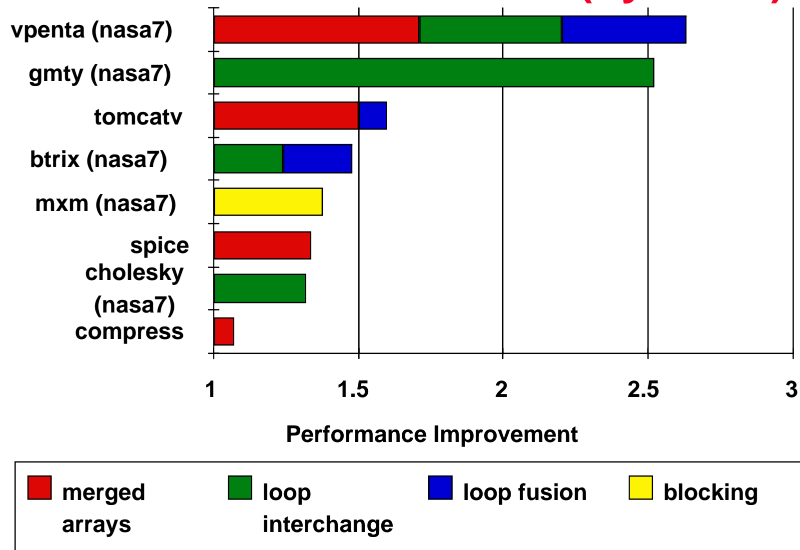
- B called ***Blocking Factor***
- Capacity Misses from $2N^3 + N^2$ to $N^3/B+2N^2$
- Conflict Misses Too?

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)



Summary: Miss Rate Reduction

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ \ accesses}{Instruction} \times \textbf{Miss rate} \times Miss\ \ penalty \right) \times Clock\ \ cycle\ \ time$$

- 3 Cs: Compulsory, Capacity, Conflict
  0. Larger cache
  1. Reduce Misses via Larger Block Size
  2. Reduce Misses via Higher Associativity
  3. Reducing Misses via Victim Cache
  4. Reducing Misses via Pseudo-Associativity
  5. Reducing Misses by HW Prefetching Instr, Data
  6. Reducing Misses by SW Prefetching Data
  7. Reducing Misses by Compiler Optimizations
- Prefetching comes in two flavors:
  - Binding prefetch: Requests load directly into register.
    - » Must be correct address and register!
  - Non-Binding prefetch: Load into cache.
    - » Can be incorrect. Frees HW/SW to guess!

Review: Improving Cache Performance

1. Reduce the miss rate,
2. *Reduce the miss penalty,* or
3. Reduce the time to hit in the cache.

Write Policy: Write-Through vs Write-Back

- Write-through: all writes update cache and underlying memory/cache
  - Can always discard cached data - most up-to-date data is in memory
  - Cache control bit: only a *valid* bit
- Write-back: all writes simply update cache
  - Can't just discard cached data - may have to write it back to memory
  - Cache control bits: both *valid* and *dirty* bits
- Other Advantages:
  - Write-through:
    - » memory (or other processors) always have latest data
    - » Simpler management of cache
  - Write-back:
    - » much lower bandwidth, since data often overwritten multiple times
    - » Better tolerance to long-latency memory?

Write Policy 2: Write Allocate vs Non-Allocate (What happens on write-miss)

- Write allocate: allocate new cache line in cache
  - Usually means that you have to do a "read miss" to fill in rest of the cache-line!
  - Alternative: per/word valid bits
- Write non-allocate (or "write-around"):
  - Simply send write data through to underlying memory/cache - don't allocate new cache line!

1. Reducing Miss Penalty: Read Priority over Write on Miss

- Write-through w/ write buffers => RAW conflicts with main memory reads on cache misses
  - If simply wait for write buffer to empty, might increase read miss penalty (old MIPS 1000 by 50%)
  - Check write buffer contents before read; if no conflicts, let the memory access continue
- Write-back want buffer to hold displaced blocks
  - Read miss replacing dirty block
  - Normal: Write dirty block to memory, and then do the read
  - Instead copy the dirty block to a write buffer, then do the read, and then do the write
  - CPU stall less since restarts as soon as do read

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

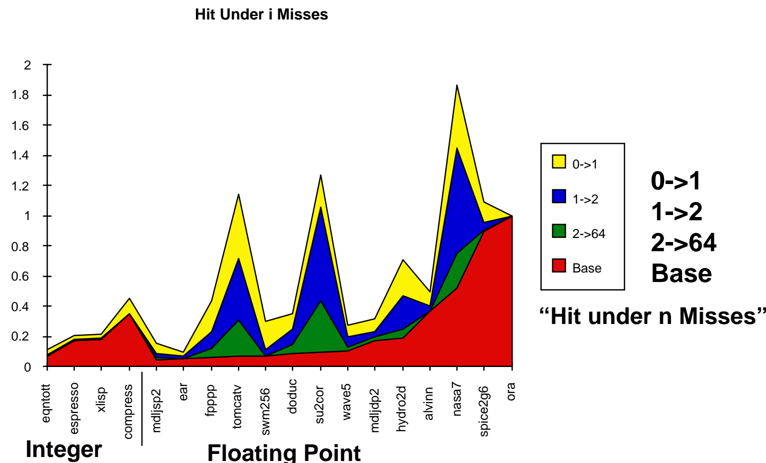
- Don't wait for full block to be loaded before restarting CPU
  - *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - *Critical Word First*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called *wrapped fetch* and *requested word first*
- Generally useful only in large blocks,
- Spatial locality => tend to want next sequential word, so not clear if benefit by early restart

block

3. Reduce Miss Penalty: Non-blocking Caches to reduce stalls on misses

- *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



- 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

4: Add a second-level cache

- **L2 Equations**

  $AMAT = Hit\ Time_{L1} + Miss\ Rate_{L1} \times Miss\ Penalty_{L1}$

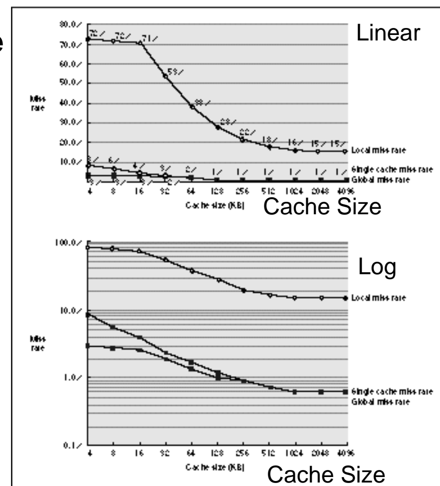
  $Miss\ Penalty_{L1} = Hit\ Time_{L2} + Miss\ Rate_{L2} \times Miss\ Penalty_{L2}$

  $AMAT = Hit\ Time_{L1} +$
  $Miss\ Rate_{L1} \times (Hit\ Time_{L2} + Miss\ Rate_{L2} + Miss\ Penalty_{L2})$

- **Definitions:**
  - *Local miss rate*— misses in this cache divided by the total number of memory accesses *to this cache* (Miss $rate_{L2}$)
  - *Global miss rate*—misses in this cache divided by the total number of memory accesses *generated by the CPU*
  - Global Miss Rate is what matters

Comparing Local and Global Miss Rates

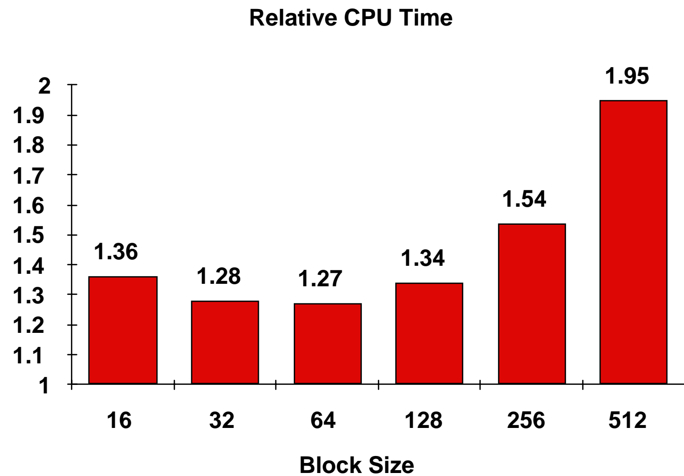
- 32 KByte 1st level cache; Increasing 2nd level cache
- Global miss rate close to single level cache rate provided L2 >> L1
- Don't use local miss rate
- L2 not tied to CPU clock cycle!
- Cost & A.M.A.T.
- Generally Fast Hit Times and fewer misses
- Since hits are few, target miss reduction



Reducing Misses: Which apply to L2 Cache?

- Reducing Miss Rate
  1. Reduce Misses via Larger Block Size
  2. Reduce Conflict Misses via Higher Associativity
  3. Reducing Conflict Misses via Victim Cache
  4. Reducing Conflict Misses via Pseudo-Associativity
  5. Reducing Misses by HW Prefetching Instr, Data
  6. Reducing Misses by SW Prefetching Data
  7. Reducing Capacity/Conf. Misses by Compiler Optimizations

L2 cache block size & A.M.A.T.



- 32KB L1, 8 byte path to memory

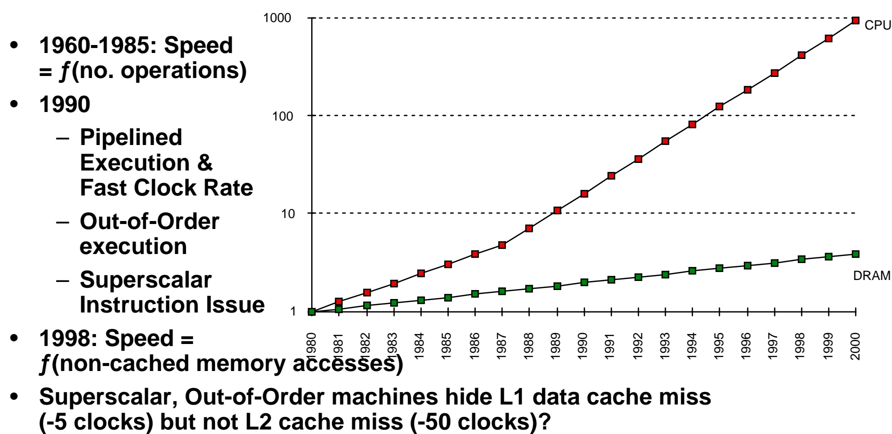
Reducing Miss Penalty Summary

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ \ accesses}{Instruction} \times \textbf{Miss rate} \times Miss\ \ penalty \right) \times Clock\ \ cycle\ \ time$$

- Four techniques
  - Read priority over write on miss
  - Early Restart and Critical Word First on miss
  - Non-blocking Caches (Hit under Miss, Miss under Miss)
  - Second Level Cache
- Can be applied recursively to Multilevel Caches
  - Danger is that time to DRAM will grow with multiple levels in between
  - First attempts at L2 caches can make things worse, since increased worst case is worse

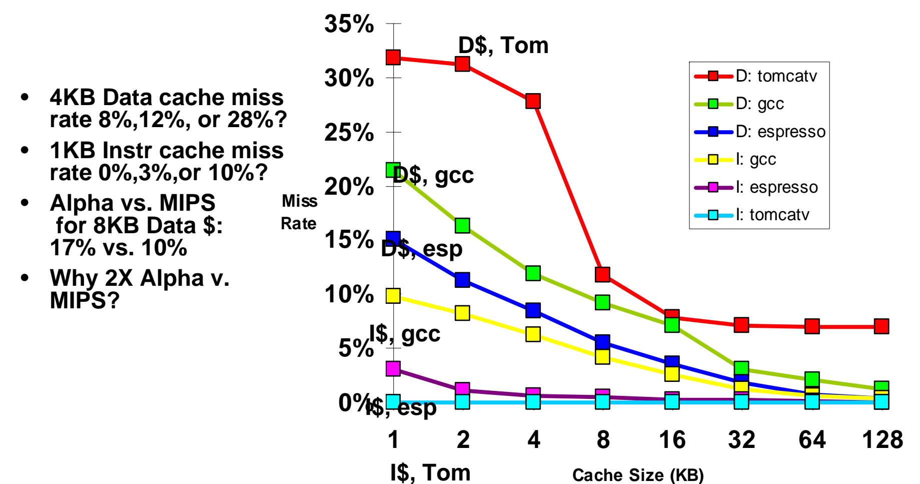
What is the Impact of What You've Learned About Caches?

- 1960-1985: Speed = ƒ(no. operations)
- 1990
  - Pipelined Execution & Fast Clock Rate
  - Out-of-Order execution
  - Superscalar Instruction Issue
- 1998: Speed = ƒ(non-cached memory accesses)
- Superscalar, Out-of-Order machines hide L1 data cache miss (-5 clocks) but not L2 cache miss (-50 clocks)?



Pitfall: Predicting Cache Performance from Different Prog. (ISA, compiler, ...)

- 4KB Data cache miss rate 8%,12%, or 28%?
- 1KB Instr cache miss rate 0%,3%,or 10%?
- Alpha vs. MIPS for 8KB Data $: 17% vs. 10%
- Why 2X Alpha v. MIPS?



Cache Optimization Summary

	Technique	MR	MP	HT	Complexity
miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Prefetching of Instr/Data	+			2
	Compiler Controlled Prefetching	+			3
	Compiler Reduce Misses	+			0
miss penalty	Priority to Read Misses		+		1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2
	Better memory system		+		3
hit time	Small & Simple Caches	–		+	0
	Avoiding Address Translation			+	2
	Pipelining Caches			+	2

Virtual Memory

The issues(s)

- DRAM is too expensive to buy gigabytes
  - Yet we want our programs to work even if they require more DRAM than we bought.
  - We also don't want a program that works on a machine with 128MB of DRAM to stop working if we try to run it on a machine with only 64MB of main memory.
- We run more than one program on the machine.

Solution: User control

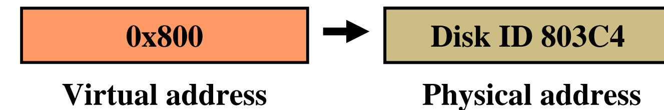
- Leave the problem to the programmer
  - Assume the programmer knows the exact configuration of the machine.
    - Programmer must either make sure the program fits in memory, or break the program up into pieces that do fit and load each other off the disk when necessary

Solution: Virtual memory

- Build new hardware and software that automatically translates each memory reference from a virtual address (that the programmer sees as an array of bytes) to a physical address (that the hardware uses to either index DRAM or identify where the storage resides on disk)

Basics of Virtual Memory

- Any time you see the word virtual in computer science and architecture it means "using a level of indirection"
- Virtual memory hardware changes the virtual address the programmer sees into the physical ones the memory chips see.



Virtual Memory View

- Virtual memory lets the programmer "see" a memory array larger than the DRAM available on a particular computer system.
- Virtual memory enables multiple programs to share the physical memory without:
  - Knowing other programs exist (transparency).
  - Worrying about one program modifying the data contents of another (protection).

Managing virtual memory

- Managed by hardware logic ***and*** operating system software.
  - Hardware for speed
  - Software for flexibility and because disk storage is controlled by the operating system.

Virtual Memory

- Treat main memory like a cache
  - Misses go to the disk
- How do we minimize disk accesses?
  - Buy lots of memory.
  - Exploit temporal locality
    - Fully associative? Set associative? Direct mapped?
  - Exploit spatial locality
    - How big should a block be?
  - Write-back or write-through?

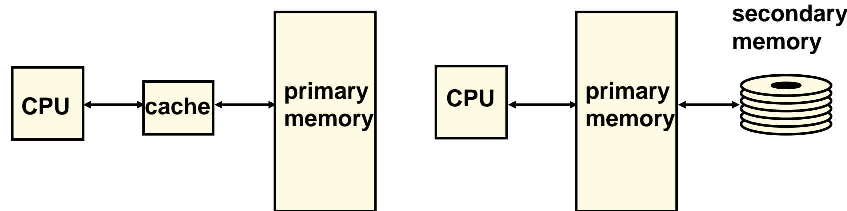
Virtual memory terminology

- Blocks are called Pages
  - A virtual address consists of
    - A virtual page number
    - A page offset field (low order bits of the address)

Virtual page number	Page offset
31	11 … 0

- Misses are called Page faults
  - and they are generally handled as an exception

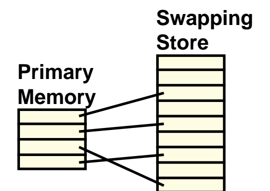
Caching vs Demand Paging



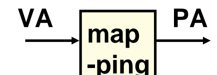
caching	demand paging
cache entry	page-frame
cache block (~16 bytes)	page (~4k bytes)
cache miss (1% to 20%)	page fault (~.001%)
cache hit (~1 cycle)	page hit (~10 cycles)
cache miss (~10 cycles)	page fault(~10K cycles)
a miss is handled in *hardware*	a miss is handled mostly in *software*

Modern Virtual Memory Systems: illusion of a large, private, uniform store

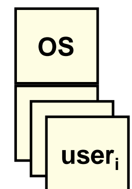
Demand Paging
capacity of secondary memory
at the speed of primary memory



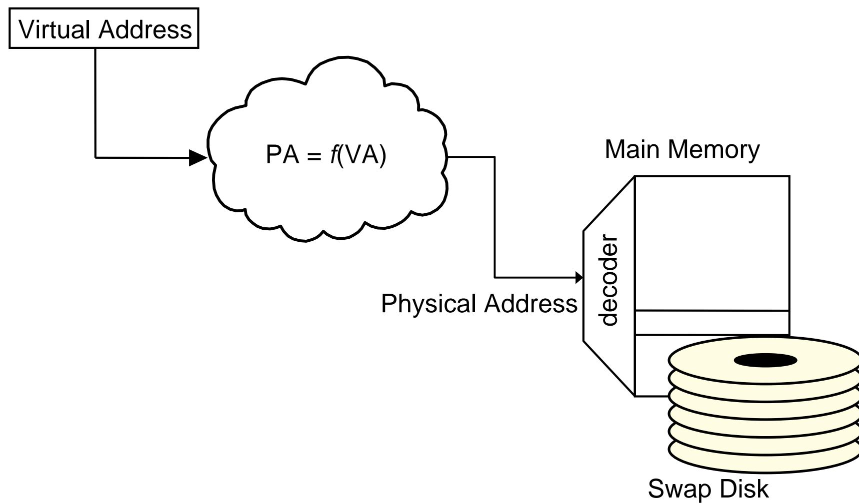
Address Translation
dynamic relocation
large “perceived” address space



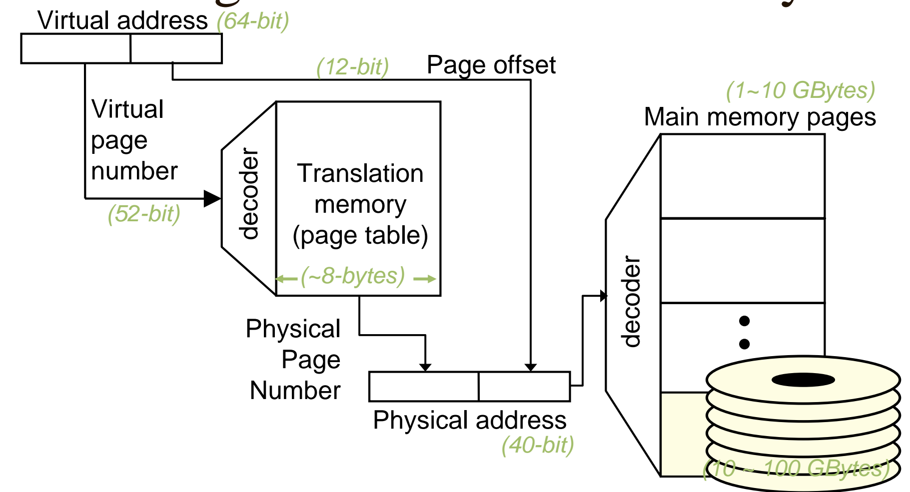
Protection
several users, each with their private address space and a common system space



Virtual to Physical Address Translation

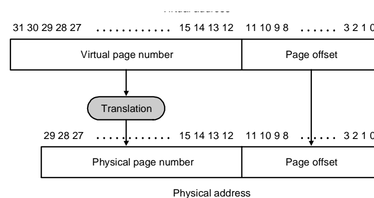


Page-Based Virtual Memory

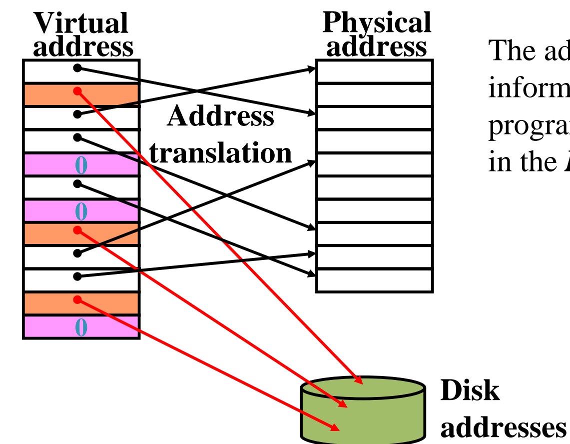


Pages: virtual memory blocks

- Page faults: the data is not in memory, retrieve it from disk
  - huge miss penalty, thus pages should be fairly large (e.g., 4KB)
  - reducing page faults is important (LRU is worth the price)
  - can handle the faults in software instead of hardware
  - using write-through is too expensive so we use writeback

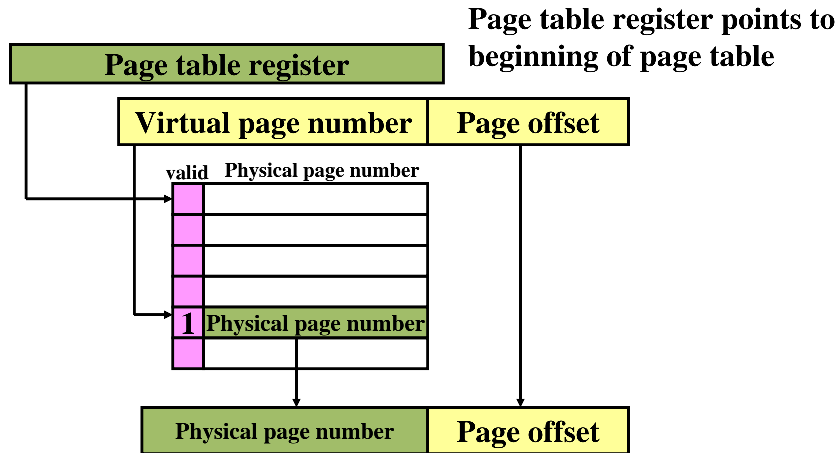


Address Translation

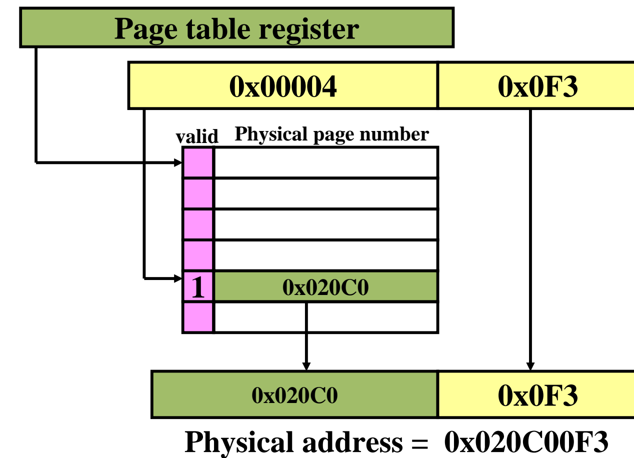


The address translation information of the program is contained in the ***Page Table.***

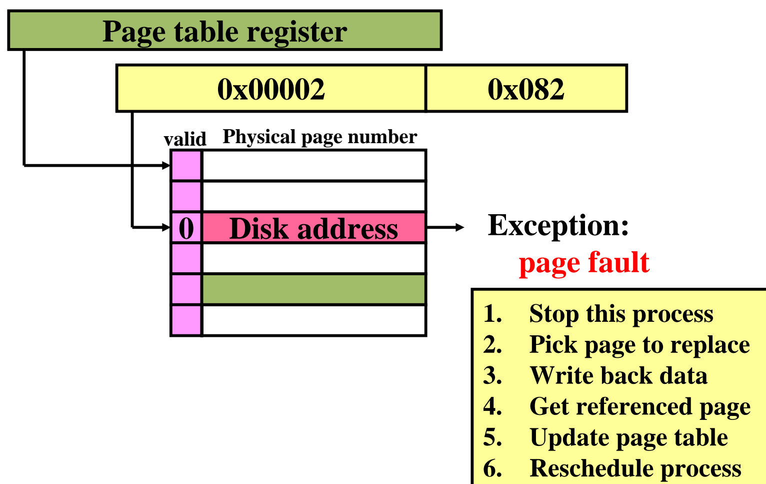
Page table components



Page table components



Page table components



How do we find it on disk?

- That is not a hardware problem! ☺
- Most operating systems partition the disk into logical devices (C: , D: , /home, etc.)
- They also have a hidden partition to support the disk portion of virtual memory
  - Swap partition on UNIX machines
  - You then index into the correct page in the swap partition.

Size of page table

- How big is a page table entry?
  - For MIPS the virtual address is 32 bits
    - If the machine can support 1GB = 2^{30} of physical memory and we use pages of size 4KB = 2^{12}, then the physical page number is 30-12 = 18 bits. Plus another valid bit + other useful stuff (read only, dirty, etc.)
    - Let say about 3 bytes.
- How many entries in the page table?
  - MIPS virtual address is 32 bits – 12 bit page offset = 2^{20} or ~1,000,000 entries
- Total size of page table: $2^{20} \times 18$ bits ~ 3 MB

Putting it all together

- Loading your program in memory
  - Ask operating system to create a new process
  - Construct a page table for this process
  - Mark all page table entries as invalid with a pointer to the disk image of the program
    - That is, point to the executable file containing the binary.
  - Run the program and get an immediate page fault on the first instruction.

Page replacement strategies

- Page table indirection enables a fully associative mapping between virtual and physical pages.
- How do we implement LRU?
  - True LRU is expensive, but LRU is a heuristic anyway, so approximating LRU is fine
  - Reference bit on page, cleared occasionally by operating system. Then pick any "unreferenced" page to evict.

Performance of virtual memory

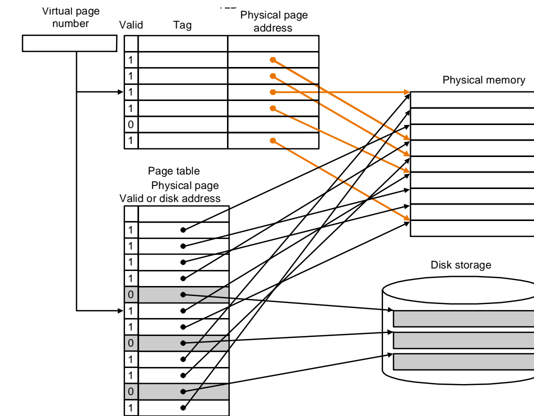
- To translate a virtual address into a physical address, we must first access the page table in physical memory.
- Then we access physical memory again to get (or store) the data
  - A load instruction performs at least 2 memory reads
  - A store instruction performs at least 1 read and then a write.
- Every memory access performs at least one slow access to main memory!

Translation lookaside buffer

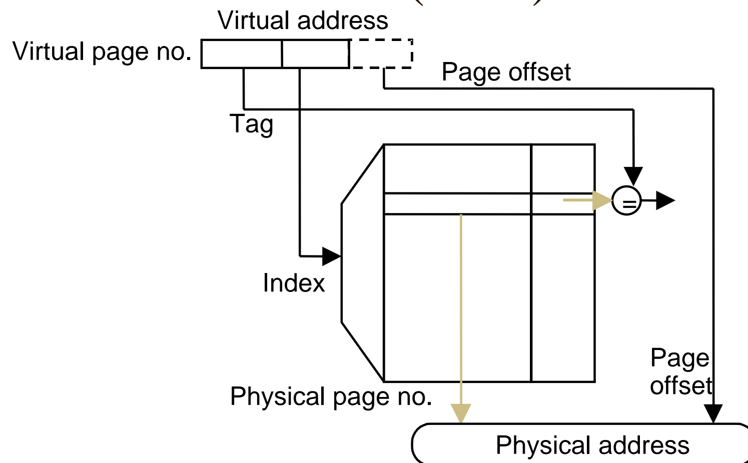
- We fix this performance problem by avoiding main memory in the translation from virtual to physical pages.
- We buffer the common translations in a **Translation lookaside buffer (TLB),** a fast cache memory dedicated to storing a small subset of valid VtoP translations.

Making Address Translation Fast

- A cache for address translations: translation lookaside buffer

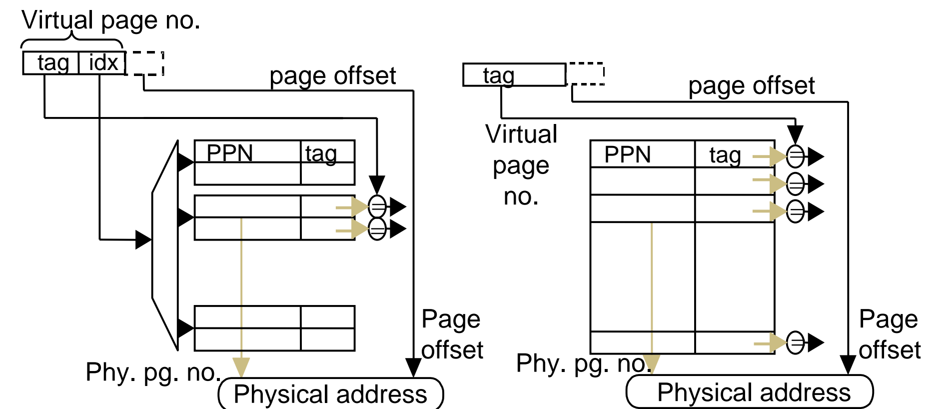


Translation Look-aside Buffer (TLB)

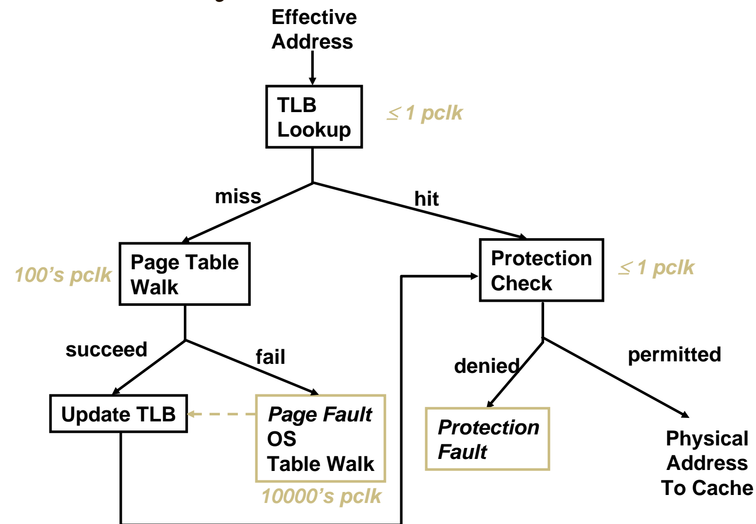


A cache of address translations

Set-Associative and Fully Associative TLBs



Virtual to Physical Address Translation



Where is the TLB lookup?

- We put the TLB lookup in the pipeline after the virtual address is calculated and before the memory reference is performed.
  - This may be before or during the data cache access.
  - Without a TLB we need to perform the translation during the memory stage of the pipeline.

Placing caches in a VM system

- VM systems give us two different addresses: virtual and physical
- Which address should we use to access the data cache?
  - Virtual address (before VM translation)
    - Faster access? More complex?
  - Physical address (after VM translations)
    - Delayed access?

Placing caches in a VM system

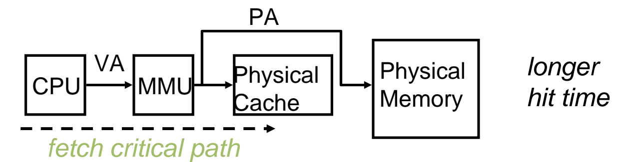
- VM systems give us two different addresses: virtual and physical
- Which address should we use to access the data cache?
  - Virtual address (before VM translation)
    - Faster access? More complex?
  - Physical address (after VM translations)
    - Delayed access?

Physically addressed caches

- Perform TLB lookup ***before*** cache tag comparison.
  - Use bits from physical address to index set
  - Use bits from physical address to compare tag
- Slower access?
  - Tag lookup takes place ***after*** the TLB lookup.
- Simplifies some VM management
  - When switching processes, TLB must be invalidated, but cache OK to stay as is.

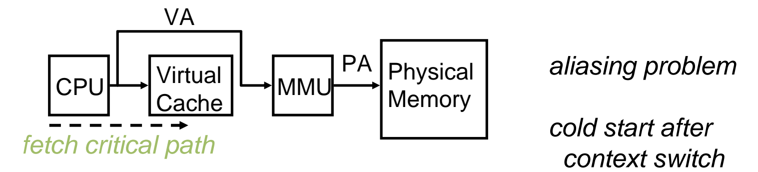
Cache Placement and Address Translation

Physical Cache (Most Systems)



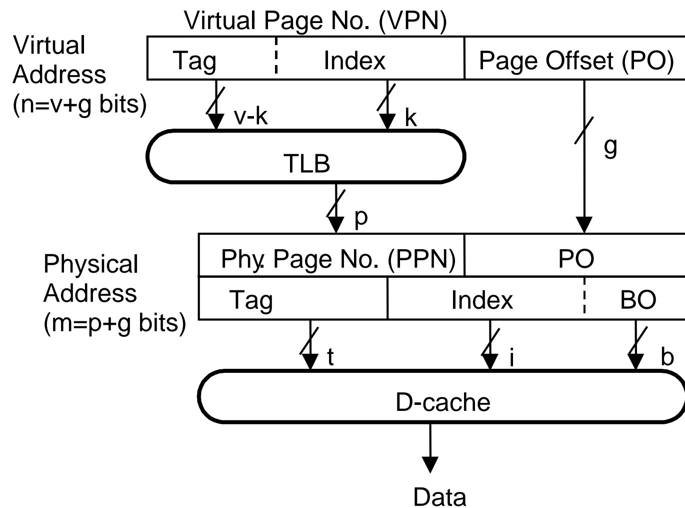
longer hit time

Virtual Cache (SPARC2's)

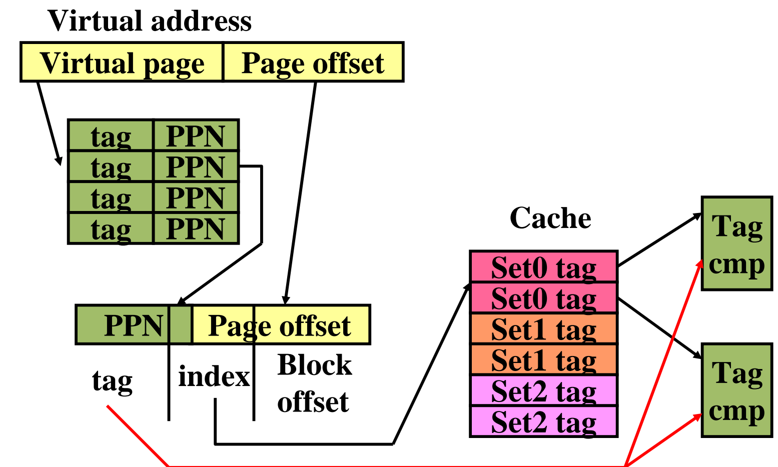


Virtual caches are not popular anymore because MMU and CPU can be integrated on one chip

Physically Indexed Cache



Picture of Physical caches



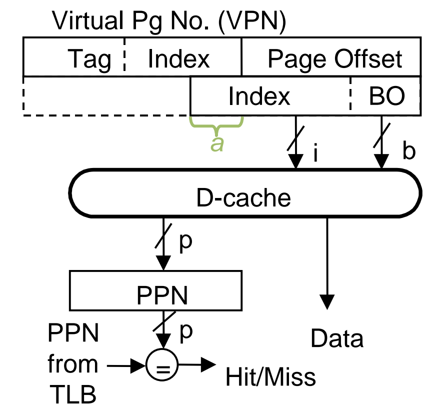
Virtually addressed caches

- Perform the TLB lookup at the same time as the cache tag compare.
  - Uses bits from the virtual address to index the cache set
  - Uses bits from the virtual address for tag match.
- Problems:
  - Aliasing: Two processes may refer to the same physical location with different virtual addresses.
  - When switching processes, TLB must be invalidated, and dirty cache blocks must be written back to memory.

Synonym (or Aliasing)

- When VPN bits are used in indexing, two virtual addresses that map to the same physical address can end up sitting in two cache lines
- In other words, two copies of the same physical memory location may exist in the cache

  ⇒ *modification to one copy won't be visible in the other*



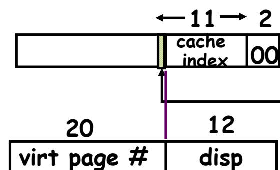
If the two VPNs do not differ in a then there is no aliasing problem

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

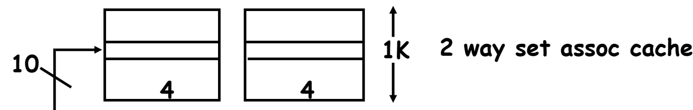
Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:



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

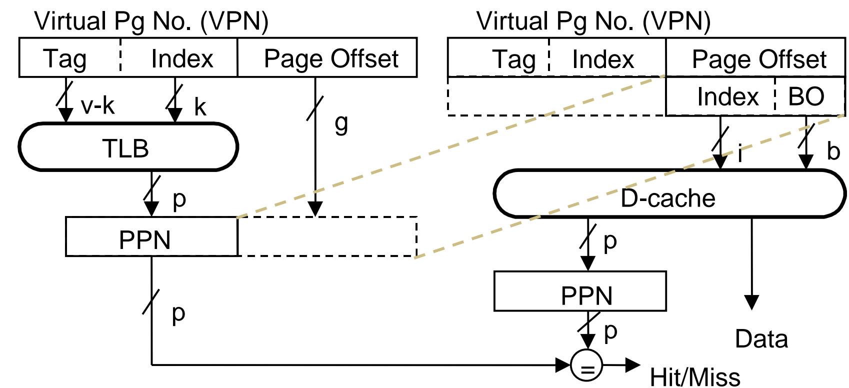
Solutions:

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



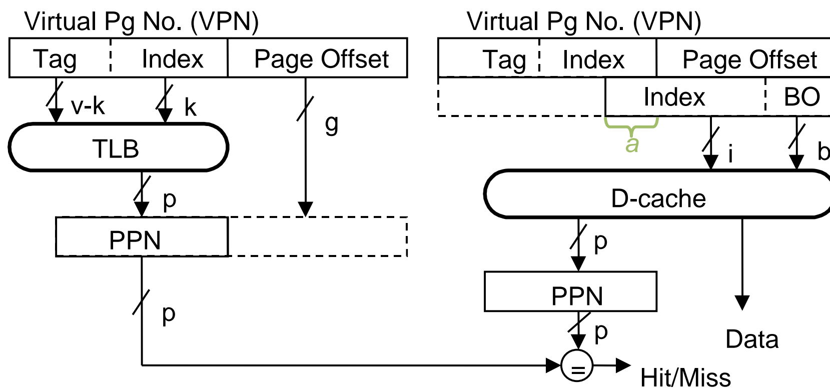
Virtually Indexed Cache

Parallel Access to TLB and Cache arrays



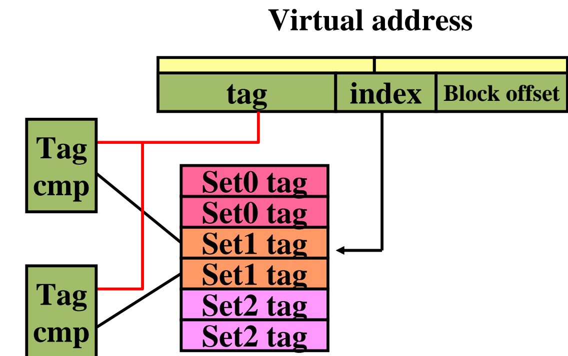
How large can a virtually indexed cache get?

Large Virtually Indexed Cache



If two VPNs differs in *a*, but both map to the same PPN then there is an aliasing problem

Picture of Virtual Caches



- **TLB is accessed in parallel with cache lookup**
- **Physical address is used to access main memory in case of a cache miss.**

OS support for Virtual Memory

- It must be able to modify the page table register, update page table values, etc.
  - To enable the OS to do this, AND not the user program, we have different execution modes for a process – one which has executive (or supervisor or kernel level) permissions and one that has user level permissions.