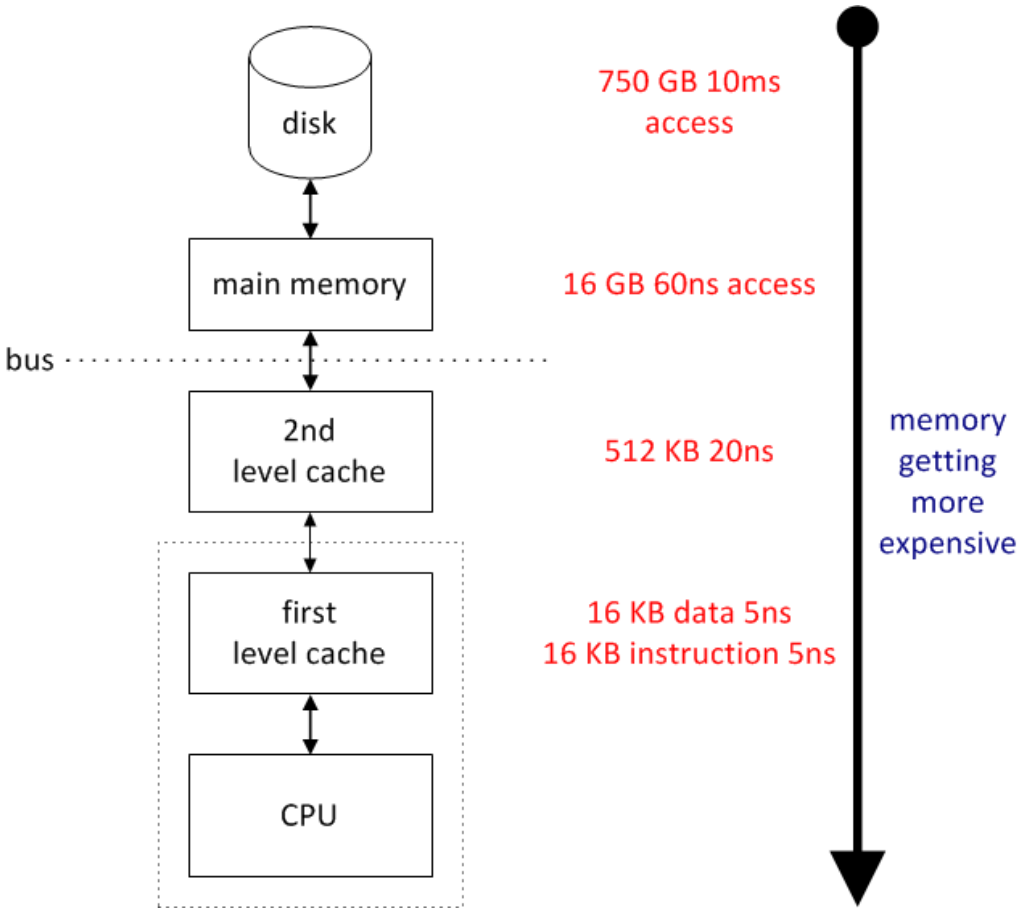
Caches

Cache Memory

- memory hierarchy
- CPU memory request presented to first-level cache first
- if data NOT in cache, request sent to next level in hierarchy…
- and so on



Cache Hierarchy

- for a system with a first level cache and memory ONLY

$t_{eff} = ht_{cache} + (1-h)t_{main}$ *where*

t_{eff} = *effective access time*
h = *probability of a cache hit [hit rate]*
t_{cache} = *cache access time*
t_{main} = *main memory access time*

- assuming t_{main} = 60ns and t_{cache} = 5ns

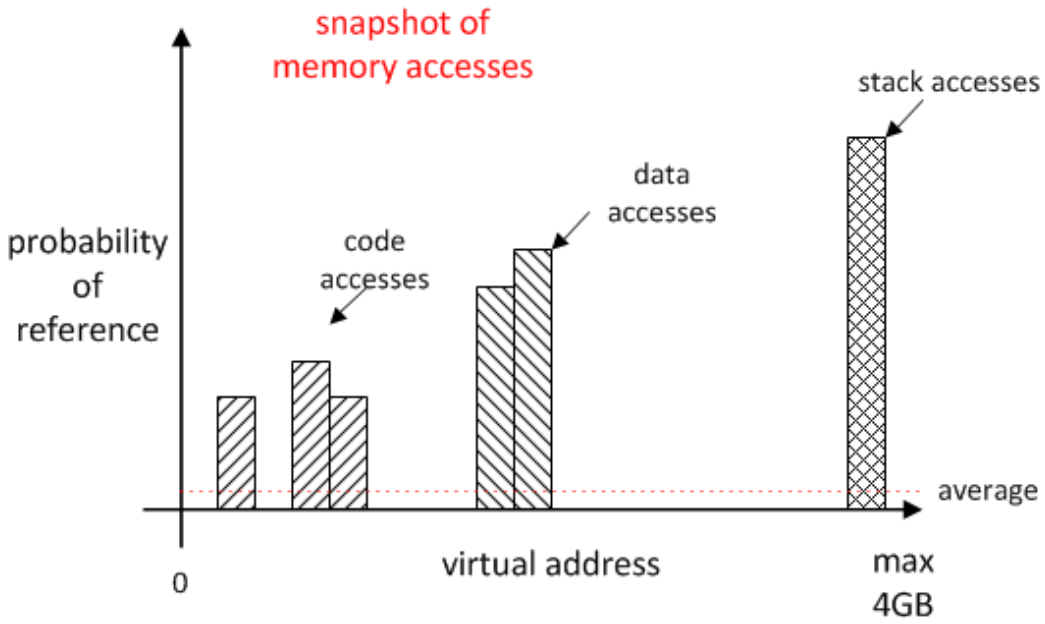
h	t_{eff} (ns)
1.00	5.0
0.99	5.6
0.98	6.1
0.89	11.1
0.50	32.5
0.00	60.0

- small changes in hit ratio [as $h \rightarrow 1$] are amplified by t_{main}/t_{cache}
- if t_{main}/t_{cache} is 10 then a decrease of 1% in h [as $h \rightarrow 1$] results in a 10% increase in t_{eff}

Temporal and Locality of Reference

- exploit the temporal locality and locality of reference inherent in *typical* programs

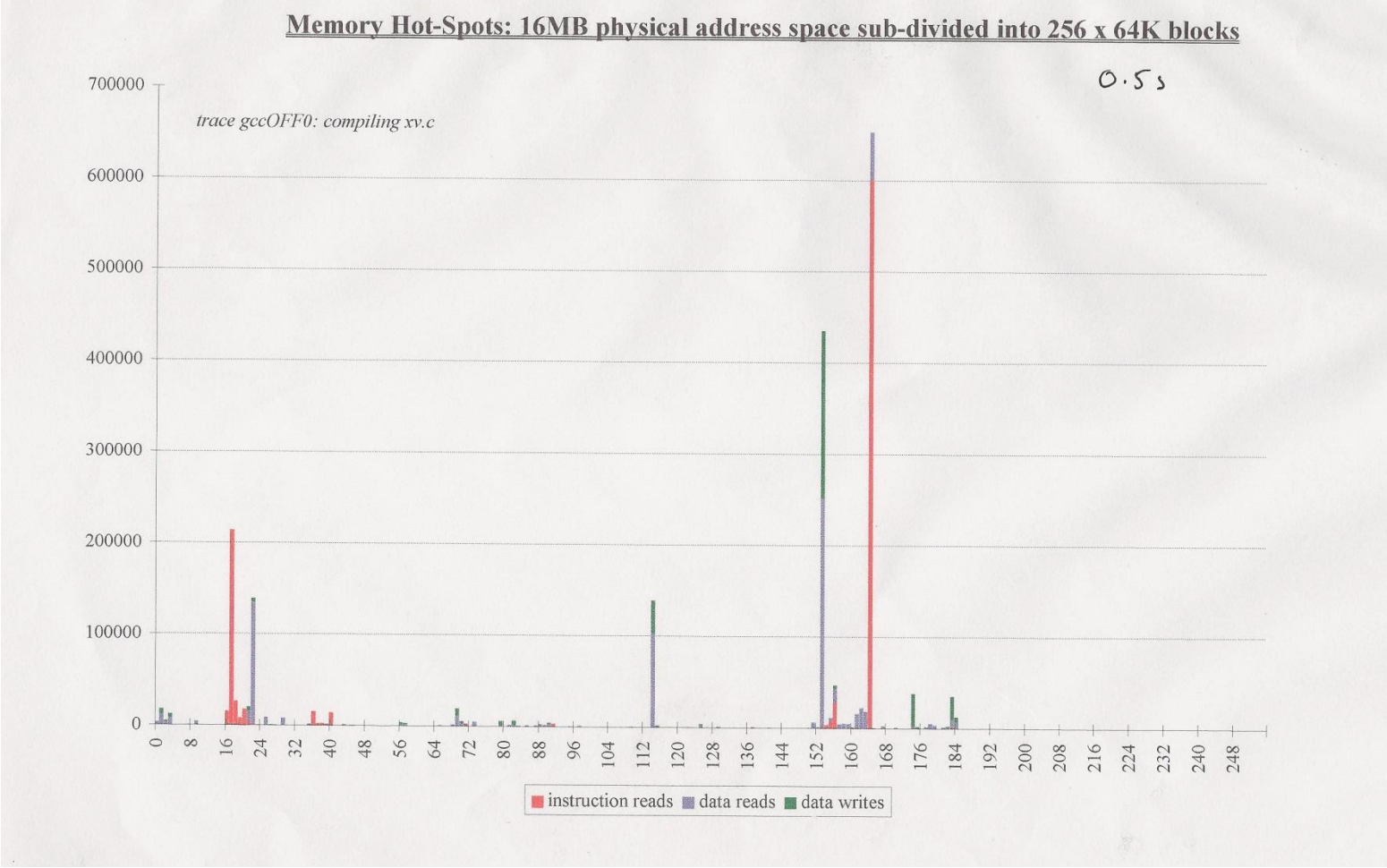
- high probability memory regions
  - recently executed code
  - recent stack accesses
  - recently accessed data



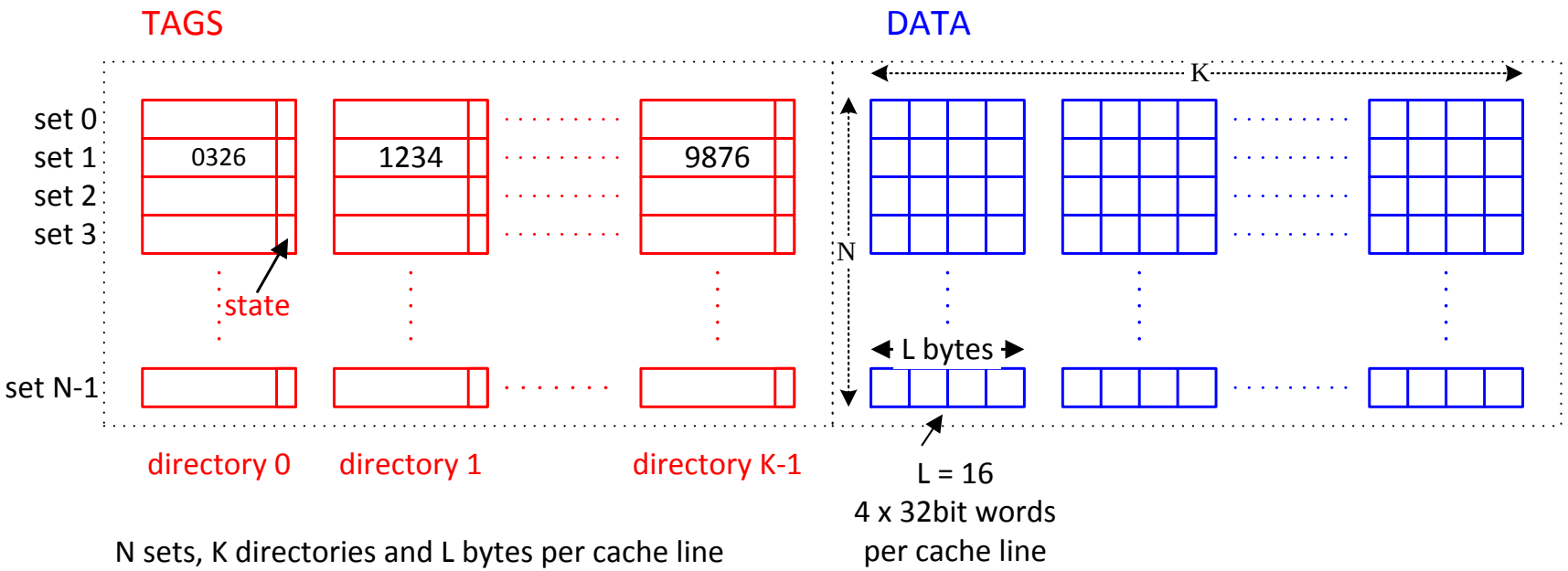
- if the memory references occur randomly, cache will have very little effect
- NB: see average on graph

Caches

A real example collected from an early IA32 PC running OSF/1



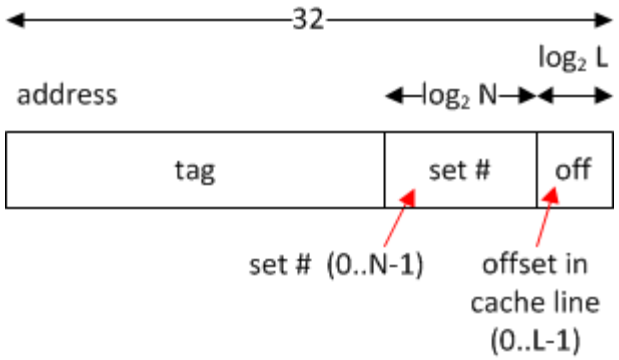
K-way Set Associative Cache with N Sets



N sets, K directories and L bytes per cache line

Caches

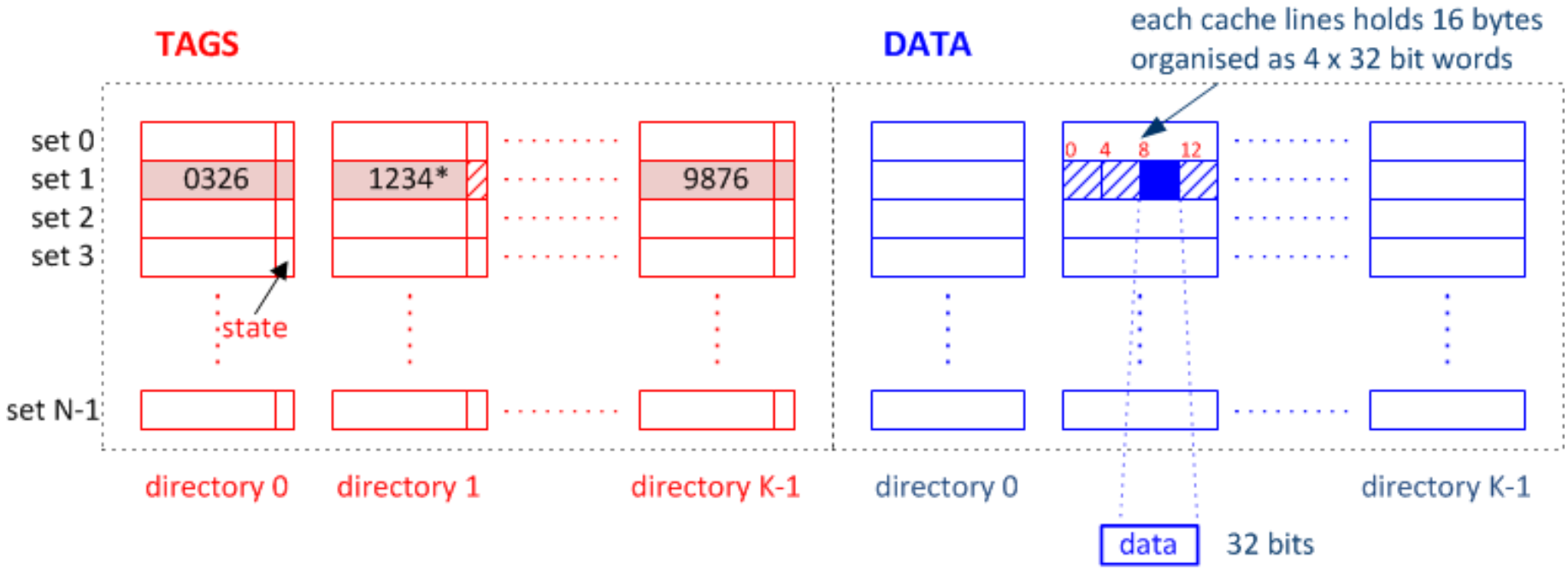
Searching a K-way Cache



- address mapped onto a particular set [set #]…
- by extracting bits from incoming address
- NB: tag, set # and offset

- consider an address that maps to set 1
- the *set 1* tags of all K directories are compared with the incoming address tag simultaneously
- if a is match found [hit], corresponding data returned offset within cache line
- the K data lines in the set are accessed concurrently with the directory entries so that on a hit the data can be routed quickly to the output buffers
- if a match is NOT found [miss], read data from memory, place in cache line within set and update corresponding cache tag [choice of K positions]
- cache line replacement strategy [within a set] - Least Recently Used [LRU], pseudo LRU, random…

Searching a K-way Cache…



Incoming address (32 bits or 8 nybbles)

1234	001	8
tag	set	off

L=16, N=4096, K = K

NB: cache lines aligned on 16 byte boundaries

Cache Organisation

- the cache organisation is described using the following three parameters

  L bytes per cache line [cache line or block size]
  K cache lines per set [degree of associativity K-way]
  N number of sets

- cache size L×K×N bytes
- N = 1
  - fully associative cache, incoming address tag compared with ALL cache tags
  - address can map to any one of the K cache lines
- K = 1
  - direct mapped cache, incoming address tag compared with ONLY ONE cache tag
  - address can be mapped ONLY onto a single cache line
- N > 1 and K > 1
  - set-associative [K-way cache]

Write-Through vs Write-Back [Write-Deferred]

- WRITE-THROUGH
  - write hit

    update cache line and main memory
  - write miss

    update main memory ONLY [non write allocate cache]

    OR

    select a cache line [using replacement policy]
    fill cache line by reading data from memory
    write to cache line and main memory [write allocate cache]

    NB: unit of writing [e.g. 4 bytes] likely to be much smaller than cache line size [e.g. 16 bytes]

Caches

Write-Through vs Write-Back [Write-Deferred]…

- WRITE-BACK [WRITE-DEFERRED]
  - write hit

    update cache line ONLY
    ONLY update main memory when cache line is flushed or replaced

  - write miss

    select a cache line [using replacement policy]
    write-back previous cache line to memory if dirty/modified
    fill cache line by reading data from memory
    write to cache line ONLY

    NB: unit of writing [e.g. 4 bytes] likely to be much smaller than cache line size [e.g. 16 bytes]

Typical Cache Miss Rates

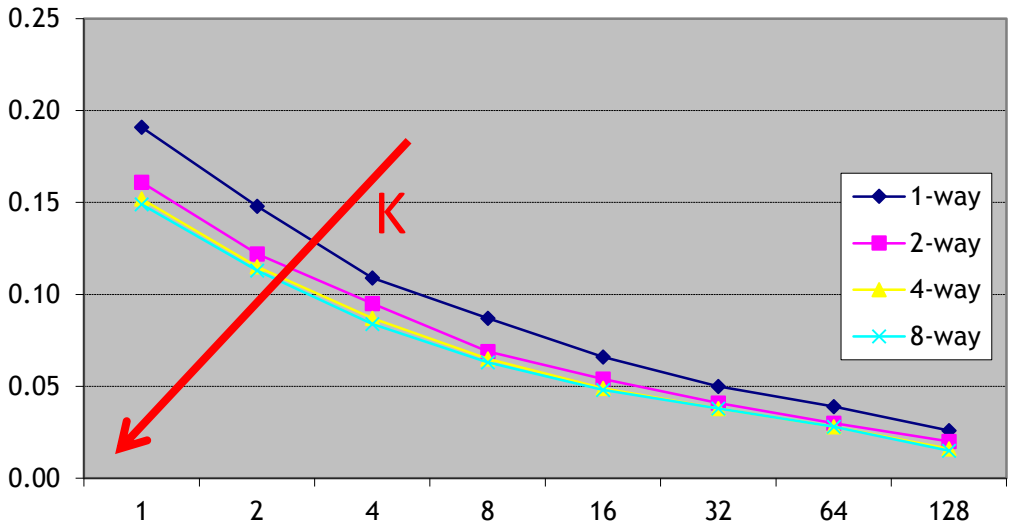
- data from *Hennessy and Patterson*
- shows *miss rate* rather than *hit rate*
- *miss rate* more interesting!
- note how data [address trace] was collected
- trace fed through a software cache model with
  - L = 32
  - LRU replacement policy

Cache size	Degree associative	Total miss rate	Miss-rate components (relative percent) (Sum = 100% of total miss rate) Compulsory		Capacity		Conflict	
1 KB	1-way	0.191	0.009	5%	0.141	73%	0.042	22%
1 KB	2-way	0.161	0.009	6%	0.141	87%	0.012	7%
1 KB	4-way	0.152	0.009	6%	0.141	92%	0.003	2%
1 KB	8-way	0.149	0.009	6%	0.141	94%	0.000	0%
2 KB	1-way	0.148	0.009	6%	0.103	70%	0.036	24%
2 KB	2-way	0.122	0.009	7%	0.103	84%	0.010	8%
2 KB	4-way	0.115	0.009	8%	0.103	90%	0.003	2%
2 KB	8-way	0.113	0.009	8%	0.103	91%	0.001	1%
4 KB	1-way	0.109	0.009	8%	0.073	67%	0.027	25%
4 KB	2-way	0.095	0.009	9%	0.073	77%	0.013	14%
4 KB	4-way	0.087	0.009	10%	0.073	84%	0.005	6%
4 KB	8-way	0.084	0.009	11%	0.073	87%	0.002	3%
8 KB	1-way	0.087	0.009	10%	0.052	60%	0.026	30%
8 KB	2-way	0.069	0.009	13%	0.052	75%	0.008	12%
8 KB	4-way	0.065	0.009	14%	0.052	80%	0.004	6%
8 KB	8-way	0.063	0.009	14%	0.052	83%	0.002	3%
16 KB	1-way	0.066	0.009	14%	0.038	57%	0.019	29%
16 KB	2-way	0.054	0.009	17%	0.038	70%	0.007	13%
16 KB	4-way	0.049	0.009	18%	0.038	76%	0.003	6%
16 KB	8-way	0.048	0.009	19%	0.038	78%	0.001	3%
32 KB	1-way	0.050	0.009	18%	0.028	55%	0.013	27%
32 KB	2-way	0.041	0.009	22%	0.028	68%	0.004	11%
32 KB	4-way	0.038	0.009	23%	0.028	73%	0.001	4%
32 KB	8-way	0.038	0.009	24%	0.028	74%	0.001	2%
64 KB	1-way	0.039	0.009	23%	0.019	50%	0.011	27%
64 KB	2-way	0.030	0.009	30%	0.019	65%	0.002	5%
64 KB	4-way	0.028	0.009	32%	0.019	68%	0.000	0%
64 KB	8-way	0.028	0.009	32%	0.019	68%	0.000	0%
128 KB	1-way	0.026	0.009	34%	0.004	16%	0.013	50%
128 KB	2-way	0.020	0.009	46%	0.004	21%	0.006	33%
128 KB	4-way	0.016	0.009	55%	0.004	25%	0.003	20%
128 KB	8-way	0.015	0.009	59%	0.004	27%	0.002	14%

FIGURE 8.12 Total miss rate for each size cache and percentage of each according to the "three Cs." Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases. Hill [1987] measured this trace using 32-byte blocks and LRU replacement. It was generated on a VAX-11 running Ultrix by mixing three systems' traces, using a multiprogramming workload and three user traces. The total length was just over a million addresses; the largest piece of data referenced during the trace was 221 KB. Figure 8.13 (page 422) shows the same information graphically. Note that the 2:1 cache rule of thumb (inside front cover) is supported by the statistics in this table: a direct-mapped cache of size N has about the same miss rate as a 2-way–set-associative cache of size N/2.

Typical Cache Miss Rates

- plot of *miss rate vs cache size* using Hennessy and Patterson data



normally

- miss rate decreases as cache size increased [or hit rate increases as cache size increased]
- miss rate decreases as K increased [or hit rate increases as K increased]

- note that the 2:1 cache rule of thumb

  "the miss rate of a direct mapped cache of size X is about the same as a 2-way set-associative cache of size X/2"

- rule supported by data [although not perfectly]

Caches

The 3 Cs

- *Hennessy and Patterson* classify cache misses into 3 distinct types
  - compulsory
  - capacity
  - conflict
- total misses = compulsory + capacity + conflict
- assume an address trace is being processed through a cache model
- compulsory misses are due to addresses appearing in the trace for the first time, the number of unique cache line addresses in trace [reduce by prefetching data into cache]
- capacity misses are the additional misses which occur when simulating a fully associative cache [reduce by increasing cache size]
- conflict misses are the additional misses which occur when simulating a non fully associative cache [reduce by increasing cache associativity K]
- see *Hennessy and Patterson* data

Direct Mapped vs Associative Caches

- will an associative cache always outperform a direct mapped cache of the same size?

- consider two caches

  K=4, N=1, L=16 [64 byte fully associative]
  K=1, N=4, L=16 [64 byte direct mapped]

  LxKxN equal…

  and the following repeating sequence of 5 addresses

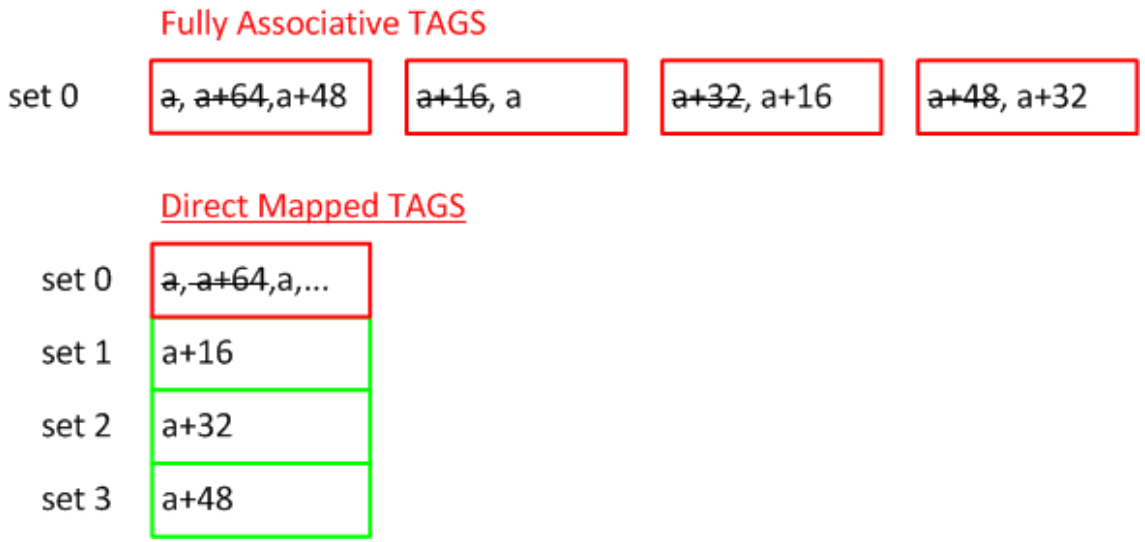
  a, a+16, a+32, a+48, a+64, a, a+16, a+32…

- increase address by 16 each time, as this is the line size [L = 16]

- caches can contain 4 addresses, sequence comprises 5 addresses
- 5 addresses won't fit into 4

Caches

Direct Mapped vs Associative Caches….



- fully associative: only 4 addresses can fit in the 4-way cache so, due to the LRU replacement policy, every access will be a miss

- direct mapped: since ONLY addresses a and a+64 will conflict with each other as they map to the same set [set 0 in diagram], there will be 2 misses and 3 hits per cycle of 5 addresses

Direct Mapped vs Associative Caches…

- the 3 Cs means that the conflict misses can be negative!
- consider previous example with 10 addresses [5 address sequence repeated twice]

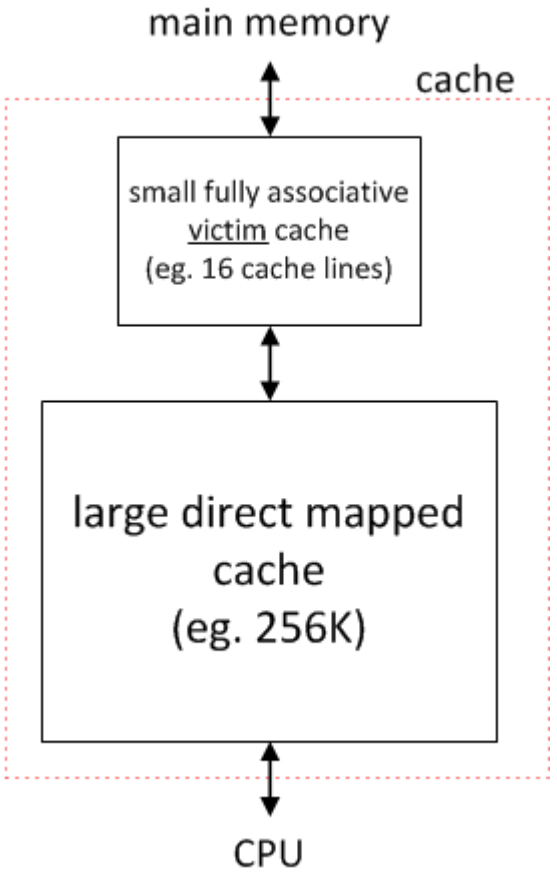
	fully associative	direct mapped
compulsory	5	5
capacity	5	5
conflict	*0*	*-3*
total	10 misses	7 misses

- calculate conflict misses from total, compulsory and capacity misses which are known
- conflict misses = total misses – compulsory misses - capacity misses
- for direct mapped cache, conflict misses = 7 – 5 – 5 = -3

Caches

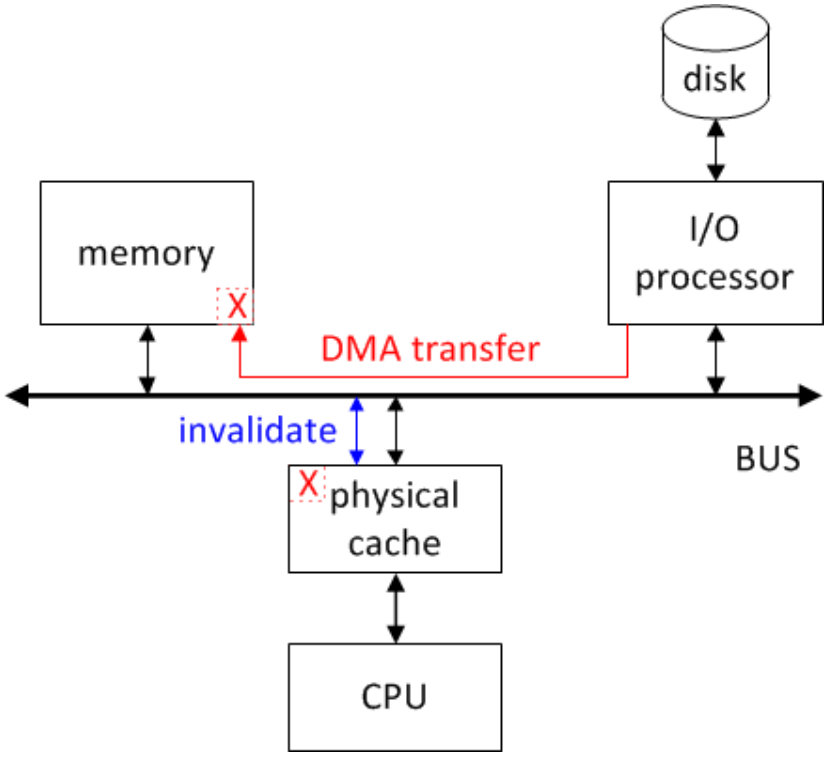
Victim Cache [Norman Jouppi]

- cost-effective cache organisation
- large direct mapped cache and a small fully-associative victim cache
- on direct-mapped cache miss, search victim cache before searching next level cache in hierarchy
- when data ejected from direct mapped cache save in victim cache
- studies indicate performance of a 2-way cache with implementation cost of a direct-mapped cache



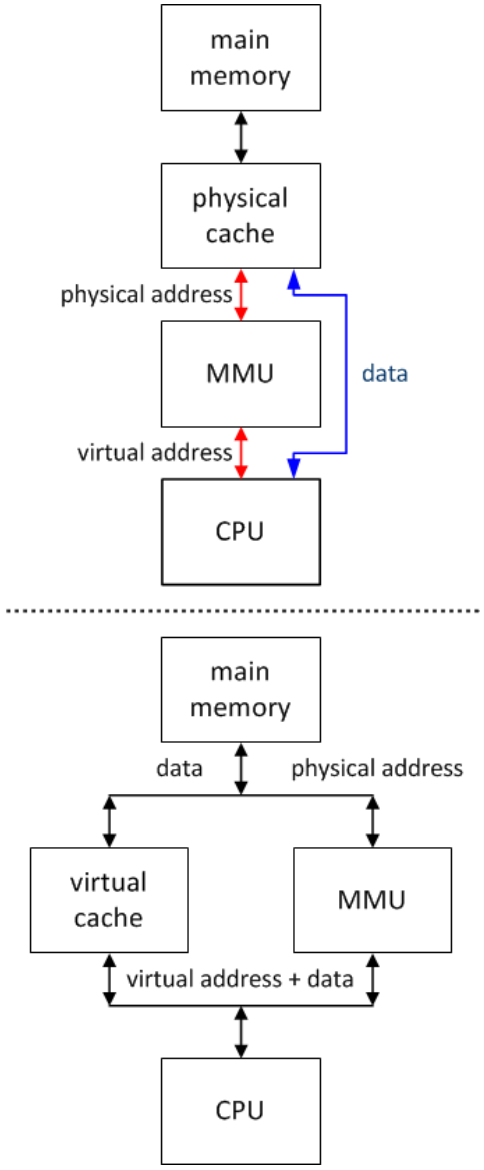
Cache Coherency

- consider an I/O processor which transfers data directly from disk to memory via an direct memory access [DMA] controller
- if the DMA transfer overwrites location X in memory, the change must somehow be reflected in any cached copy
- the cache watches [snoops on] the bus and if it observes a write to an address which it has a copy of, it invalidates the appropriate cache line [invalidate policy]
- the next time the CPU accesses location X, it will fetch the up to date copy from main memory



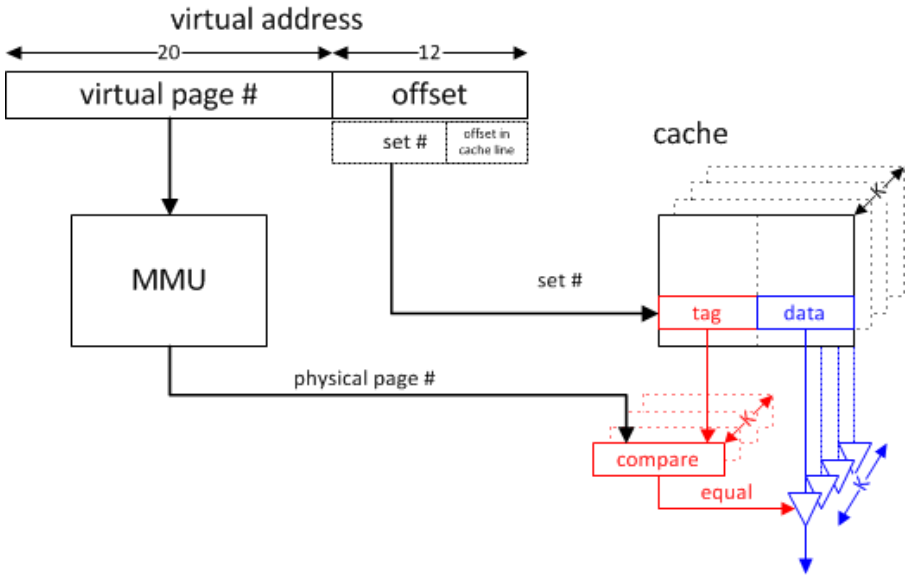
Virtual or Physical Caches?

- can both be made to work?
- possible advantages of virtual caches
  - speed? (i) no address translation required before virtual cache is accessed and (ii) the cache and MMU can operate in parallel [will show later that this advantage is not necessarily the case]
- possible disadvantages of virtual caches
  - aliasing [same problem as TLB], need a process tag to differentiate virtual address spaces [or invalidate complete cache on a context switch]
  - process tag makes it harder to share code and data
  - on TLB miss, can't walk page tables and fill TLB from cache
  - more difficult to maintain cache coherency?



A Fast Physical Cache

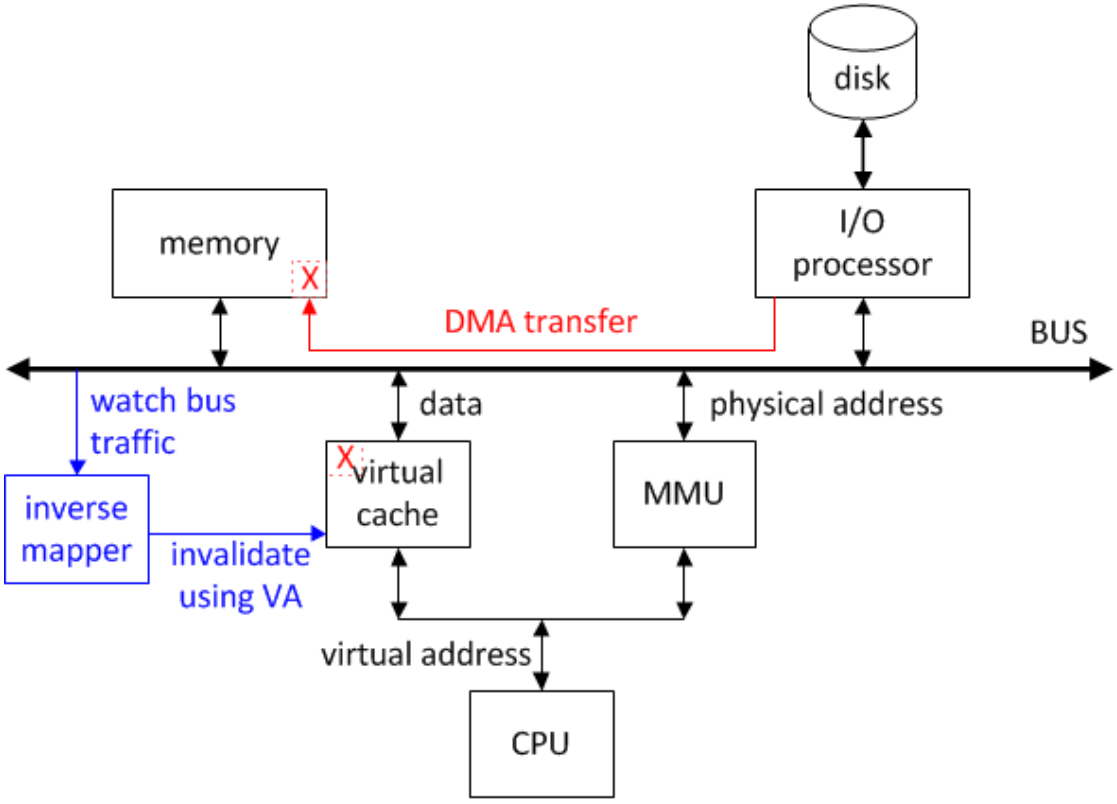
- organisation allows concurrent MMU and cache access [as per virtual cache]



- cache look-up uses low part of address which is NOT altered by the MMU
- K directories, K comparators and K buffers needed for a K-way design
- cache size = K x PAGESIZE [if L = 16, N = 256]
- *negligible* speed disadvantage compared with a virtual cache

Cache Coherency with a Virtual Cache

- address stored in cache by virtual address, but addresses on bus are physical
- need to convert physical address on bus to a virtual address in order to invalidate appropriate cache line
- could use an inverse mapper [as in diagram]



Cache Coherency with a Virtual Cache…

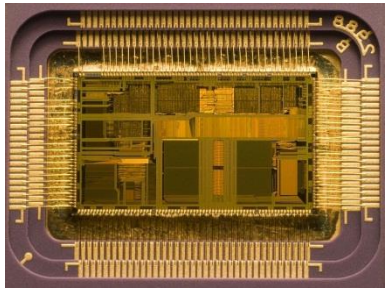
- alternatively store a physical and a virtual tag for each cache line

set 0	physical tag	virtual tag	data
set 1	physical tag	virtual tag	data
set n	physical tag	virtual tag	data

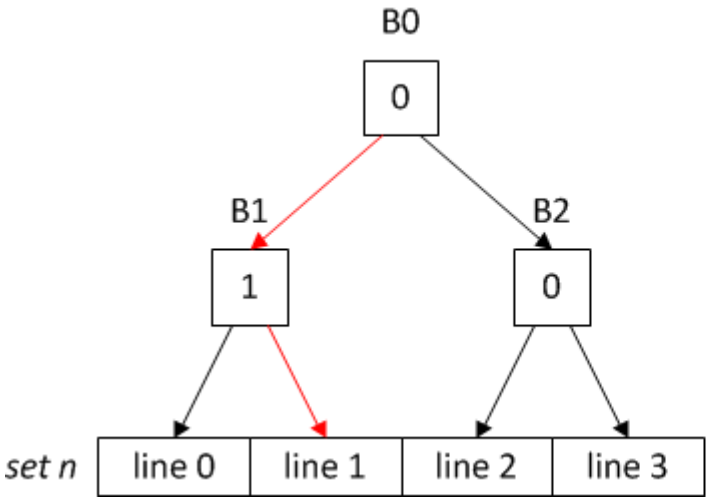
- CPU accesses match against virtual tags
- bus watcher accesses match against physical tags
- on a CPU cache miss, virtual and physical tags updated as part of the miss handling
- cache positioned between CPU and bus, needs to *look* in two directions at once [think rabbit or chameleon which has a full 360-degree arc of vision around its body]
- even with a physical cache, normal to have two identical physical tags
  - one for CPU accesses and one for bus watching

Caches

Intel 486 [1989]



- 8K physical unified code and data cache
- write-through, non write allocate
- 4-way set associative 16 bytes per line L=16, K= 4 hence N=128 [a fast physical cache]
- pseudo LRU



B2	B1	B0

(K-1) bits per set

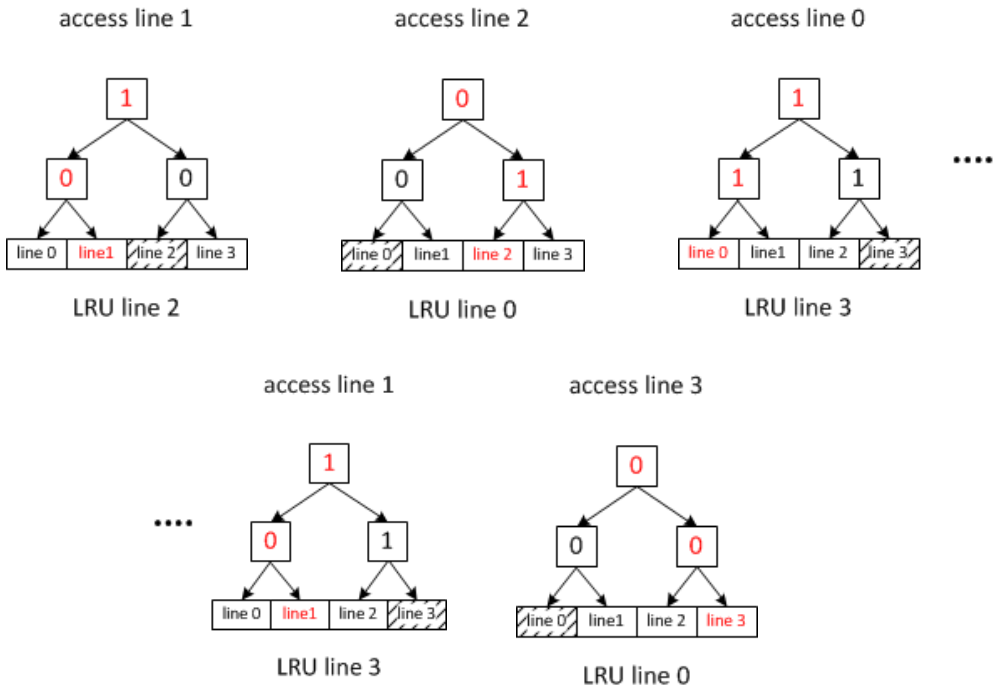
to find pseudo LRU line *go left* if bit == 0

on access set bits in tree to point away from accessed cached line

LRU is line 1

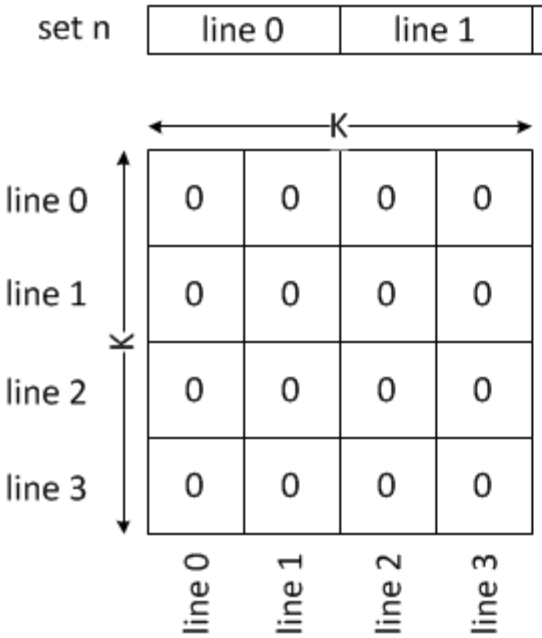
Pseudo-LRU access sequence

- consider line accesses made in following order 1, 2, 0, 1, 3
- assume pseudo LRU bits initially 0



Implementing Real LRU

- method due to Maruyama [IBM]
- keep a K^2 matrix of bits for each set



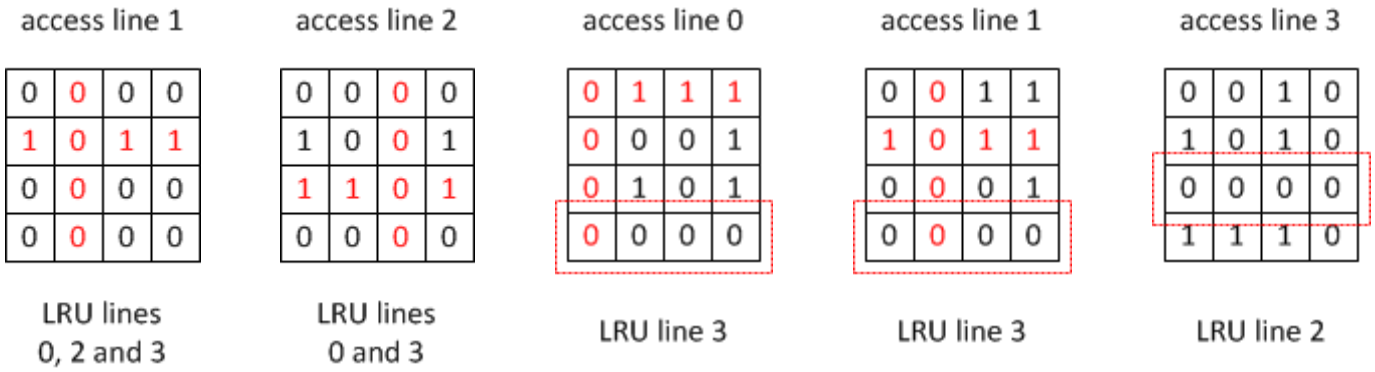
when line accessed

(i) set corresponding row to ALL ones
(ii) set corresponding column to ALL zeroes

LRU line(s) corresponds to ALL zero row

Implementing Real LRU…

- consider line accesses made in following order 1, 2, 0, 1, 3



- line 2 is LRU after access sequence

- K-1 bits per set for pseudo LRU
- K^2 bits per set for real LRU

Caches

Intel 486 [1989]…

- TLB
  - 32 entry fully associative, pseudo LRU
- non-cacheable I/O devices [e.g. polling a serial interface]
  - will not see changes if always reading cached copy [volatile]
  - can set bit in PTE to indicate that page is non-cacheable

    OR…
  - assert hardware signal when accessed to indicate that memory access should be treated as non-cacheable

Caches

Intel Core i7-3820 CPU @ 3.60GHz [Q1 2012]

- 32nm, 4 core [8 threads], L1, L2 and L3 line size 64 bytes

- L1 instruction 32K 8-way write-through per core [fast physical cache]
- L1 data 32K 8-way write-back per core [fast physical cache]
- L1 cache latency 3 clock cycles

- L2 256KB 8-way write-back unified cache per core [fast physical cache]
- L2 cache latency 12 clock cycles

- L3 10MB 20-way write-back unified cache shared by ALL cores [fast physical cache]
- L3 cache latency 26-31 clock cycles

- L1 instruction TLB , 4K pages, 64 entries, 4-way
- L1 data TLB, 4K pages, 64 entries, 4-way

- L2 TLB, 4K pages, 512 entries, 4-way

- ALL caches and TLBs use a pseudo LRU replacement policy

Cache Trace Analysis

- empirical observations of typical programs has produced the simple 30% rule of thumb:

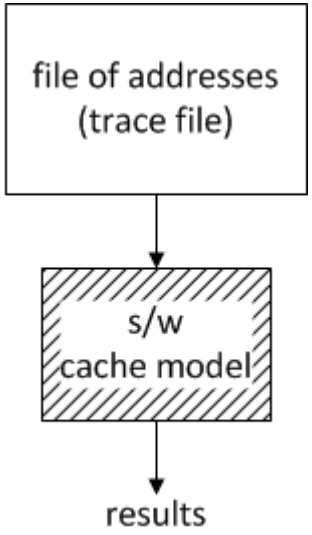
  "*each doubling of the size of the cache reduces the misses by 30%*"

- good for rough estimates, but a proper design requires a thorough analysis of the interaction between a particular machine architecture, expected workload and the cache design

- some methods of address trace collection:

  - logic analyser [normally can't store enough addresses]
  - s/w machine simulator [round robin combination of traces as described in Hennessy and Patterson]
  - instruction trace mechanism
  - microcode modification [ATUM]

- ALL accesses [including OS] or application ONLY
- issue of quality and quantity



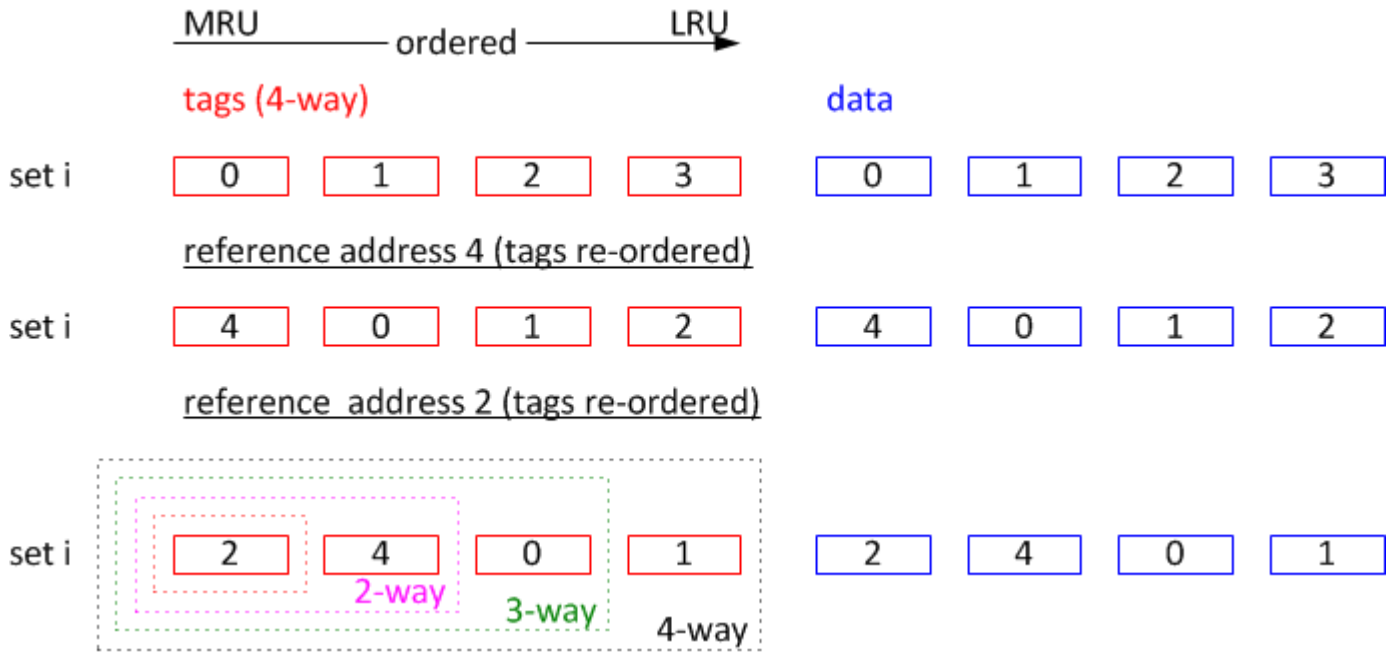
Caches

Trace File Size

- how many addresses are required to obtain statistically significant results?
- must overcome initialisation transient during which the *empty* cache is filled with data
- consider a 32K cache with 16 bytes per line => 2048 lines
  - to reduce transient misses to less than 2% of total misses, must generate at least 50 x transient misses [50 x 2048 ≈ 100,000] when running simulation
  - if the target miss ratio is 1% this implies 100,000 x 100 ≈ 10 million addresses
- evaluating N variations of cache a design on separate passes through a large trace file could take reasonable amount of CPU time
- will examine some techniques for reducing this processing effort
- in practice, it may no longer be absolutely necessary to use these techniques, but knowledge of them will lead to a better understanding of how caches operate [eg can analyse 2 million addresses in 20ms on a modern IA32 CPU]

Multiple Analyses per run

- if the cache replacement policy is LRU then it is possible to evaluate all k-way cache organisations for k < K during a single pass through the trace file



4-way cache directory (for one set) maintained with a LRU policy

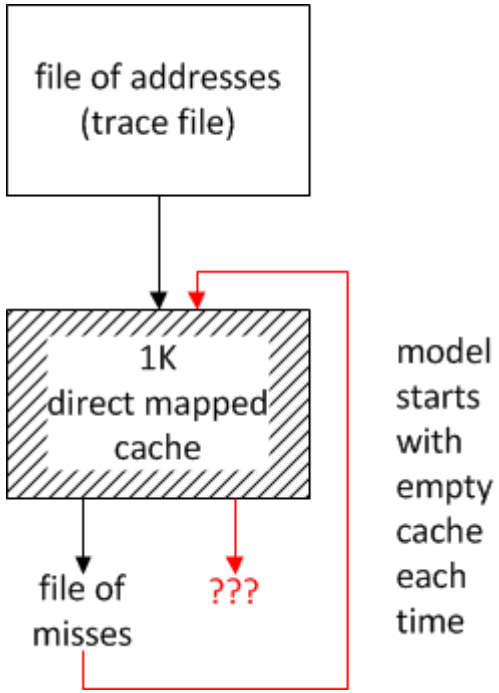
Caches

Multiple Analyses per run…

- to keep track of the hits of a 1-way to a K-way cache must simply note the position of each hit in the cache directory
- keep a vector of hit counts int hits[K]
- if a hit occurs at position i then increment hits[i]
- Increment hits for directory[0] in hits[0], directory[1] in hits[1], …
- to find the hits for a k-way cache simply sum hits[i] for i = 0 to k-1
- NB: as k increases so does the cache size
- NB: collecting hits for 1K 1-way, 2K 2-way, 3K 3-way, 4K 4-way, …

Trace Stripping

- generate a reduced trace by simulating a 1-way cache with N sets and line size L, outputting only those addresses that produce misses
- reduced trace ≈20% the size of full trace [see *Hennessy and Patterson* table for miss rate of a 1K 1-way cache]
- what can be done with the reduced trace?
- since it's a direct mapped cache, a hit doesn't change the state of the cache [no cache line tags to re-order]
- all the state changes are recorded in the file of misses
- simulating a k-way cache with N sets and line size L on the full and reduced traces will generate the same number of cache misses [simple logical argument]
- NB: as k increases so does the cache size [again]

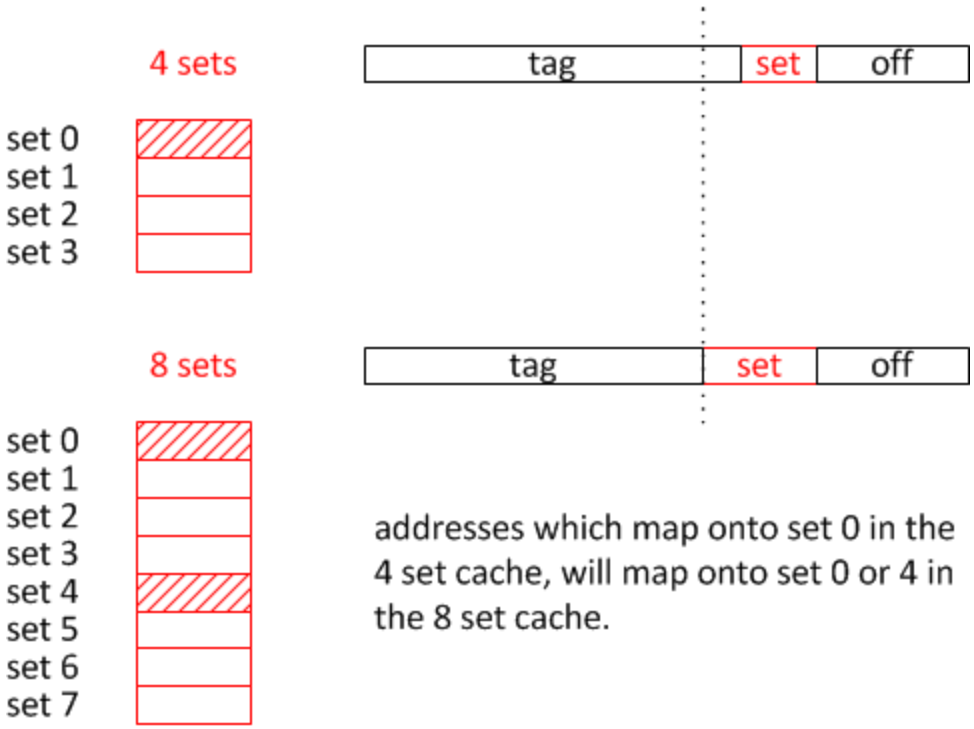


??? identical to *file of misses*
what goes in come out!

Caches

Trace Stripping…

- not only can k be varied on the reduced trace but also N in multiples of 2
- consider a reduced trace generated from a 1-way cache with 4 sets



Caches

Trace Stripping…

- reduced trace will contain addresses where the previous set number is identical, but the previous least significant tag bit is different
- this means that all addresses that change set 0 and set 4 will be in the reduced trace
- hence any address causing a miss on the 8 set cache is present in the reduced trace
- can reduce trace further by observing that each set behaves like any other set
- Puzak's experience indicates that for reasonable data, retaining only 10% of sets [at random] will give results to within 1% of the full trace 95% of the time
- see *High Performance Computer Architecture* Harold S. Stone for more details

Summary

- you are now able to
  - explain why caches work
  - explain the organisation and operation of caches
  - calculate hits, misses and the 3 Cs given an address trace and cache organisation
  - know the difference between virtual and physical caches
  - explain how LRU and pseudo LRU replacement algorithms are implemented
  - write a cache simulation
  - use a number of techniques to speed up cache simulations