Contents

Cache Memory

Memory Hierarchy

- Disk
- Main memory
- 2nd level cache
- 1st level cache
- CPU

Check is each level for the memory

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 miss
  - t_{cache} = cache access time
  - t_{main} = main memory access time
- Small changes in hit ratio are camplified by $tfrac_{main}t_{cache}$

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

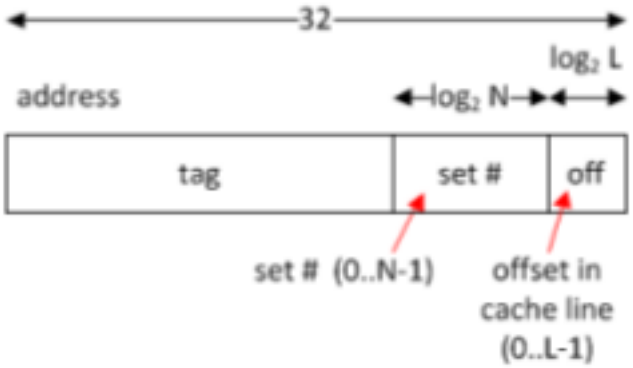

Figure 1: Address

Searching a K-way Cache

- Address mapped onto a particular set by exracting bits from incoming address
- Consider an address that maps to set 1
- The *set 1* tags of all *K directories* are compared with the incoming address tag simultaneously
  - Each set is compared against in K directories
  - I.e. If only one set exists, any data blocked can be mapped to any cache block frame
- If a match is 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…

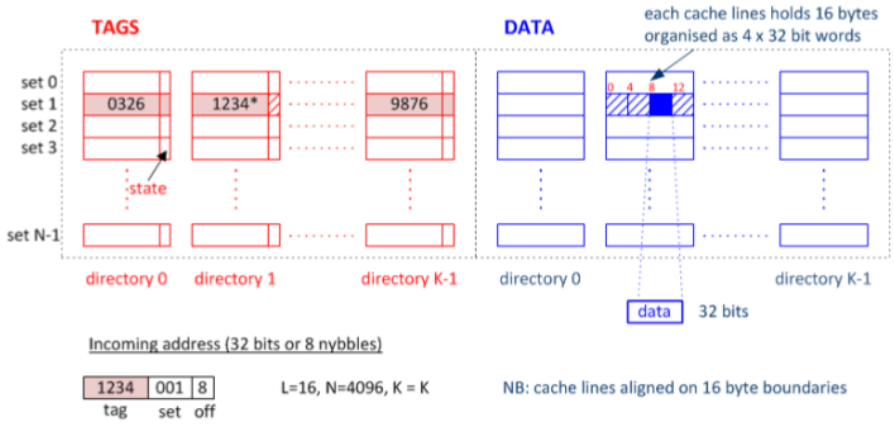

Figure 2: Structure

Cache Organisation

- The cache organisation is described using the following three parameters
  - L - bytes per cache line
  - N - number of sets
  - K - cache lines per set
- 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 (All copies updated right away)
  - Write hit
    * Update cache line and main memory
  - Write miss
    * Update main memory only (non write allocate cache)
    * or
    * Select a cache line
    * Fill cache line by reading data from memory
    * Write to cache line and main memory (write allocate cache)
- Write back (write deferred - write to main memory when cache entry is evisted, or pending changes are made)
  - Write hit
    * Update cache line ONLY
    * *ONLY update main memory when cache line is flushed or replaced*
  - Write miss
    * Select a cache line
    * *write-back previous cache line to memory if dirty/modified*
    * Fill cache line by reading data from memory
    * Write to cache line ONLY

Typical Cache Miss Rates

- Data from *Hennessy and Patterson*
- Miss rate decreases as cache size increased
- Miss rate descreases as K increased

  "This 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"

The 3 Cs

- *Hennessy and PAtterson* classify cache misses into 3 distinct types
  - Compulsory
  - Capacity
  - Conflict
- Total missed = 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 address 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)
  - A reference was stored in the cache but was evicted when there wasn't enough space
- *Conflict*
  - Misses are the additional misses which occur when simulating a non fully associate cache (reduce by increasing cache associativity K)
  - A reference was stored in the cache but was evicted when when there was another reference to another block in the block frame's set
  - These are misses that *would not occur if the cache were fully associative with LRU replacement*

Direct Mapped vs Associative Caches

- Will an associative cache *always* outperform a direct mapped cache of the same size?
  - Fully Associative
    * K=4
    * N=1
    * L=16
  - Direct Mapped
    * K=1
    * N=4
    * L=16
- Given addresses `a`, `a+16`, `a+32`, `a+48`, `a+64`, `a`, …
- Increase by 16 each time as L=16 (sets picked from bits 5 and 6)
- Fully associative
  - Only 4 addresses can fit in the 4-way cache
  - 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
  - There will be 2 misses and 3 hits per cycle of 5 addresses

Victim Cache (Normal 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 performed 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 a 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 access 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?
    * No address translation rquired before virtual cache is accessed
    * The cache and MMU can operate in parallel
- Possible disadvantages of virtual caches
  - Aliasing (same problem as translation lookaside buffer), 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?

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 × 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
- Alternatively store a physical and a virtual tag for each cache line
- 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 (this 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

Intel 486

- 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=123 (a fast physical cache)
- Pseudo LRU
  - Go left if bit==0
  - On access set bits in tree to point after from accessed cached line
  - LRU is where bits in the tree point

Example

- Consider line access made in the following order 1, 2, 0, 1, 3

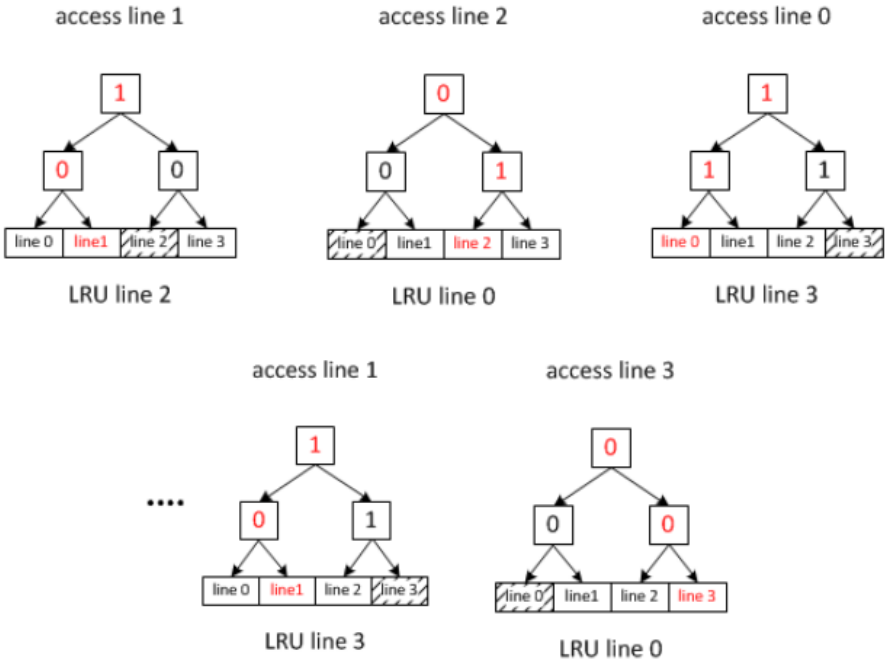

Figure 3: Pseudo-LRU

Implementing Real LRU

- Method due to Maruyama (IBM)
- Keep a K^2 matrix of bits for each set

Set n:

line 0	line 1	line 2	line 3

line0	0	0	0	0
line 1	0	0	0	0
line 2	0	0	0	0
line 3	0	0	0	0

- When line is accessed
  - Set correponding row to ALL ones
  - Set corresponding column to ALL zeroes
- LRU line(s) corresponds to ALL zero row

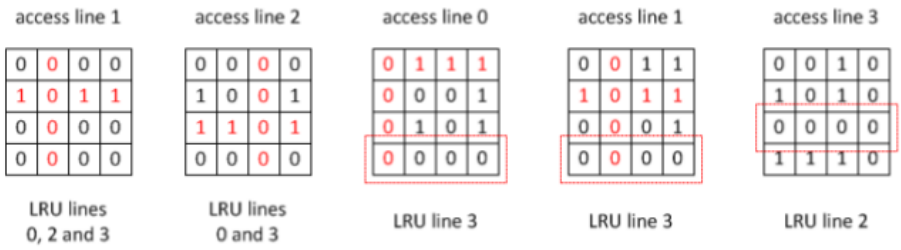

Figure 4: Implementing LRU

- 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 on 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

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 missed 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

  - s/w machine simulator
  - instruction trace mechanism
  - microcode modification
- All access or application ONLY
- Issue of quality and quantity

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 byte per line $\Rightarrow$ 2048 lines
  - To reduce transient misses to less than 2% of total misses, must generate at least 50 $\times$ transient misses when running simulation
  - If the marget miss ratio is 1% this implies 100,000 $\times$ 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

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
- 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]`
- Increments 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 address that produce misses

- Reduced trace by about 20% the size of full trace
- 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
- 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
- NB: as k increases so does the cache size
- 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
- Reduced trace will contain address where the previous set numbr is identical, but the previous least significant tag bit is different
- This means that all addresses that change set 0 and set 4 wil 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