Memory Systems

Basic caches

- introduction
- fundamental questions
- cache size, block size, associativity

Advanced caches

Prefetching

Motivation

CPU can go only as fast as memory can supply data

Assume 3Ghz CPU = 333ps cycle

4 insts per cycle = 4 references per cycle

30% is reads -> 1.2 references per cycle

total 5.2 references per 330ps

Assume 4 byte refs = 20.8 bytes per cycle

Bandwidth demand?

(even with ~2 IPC bandwidth is high)

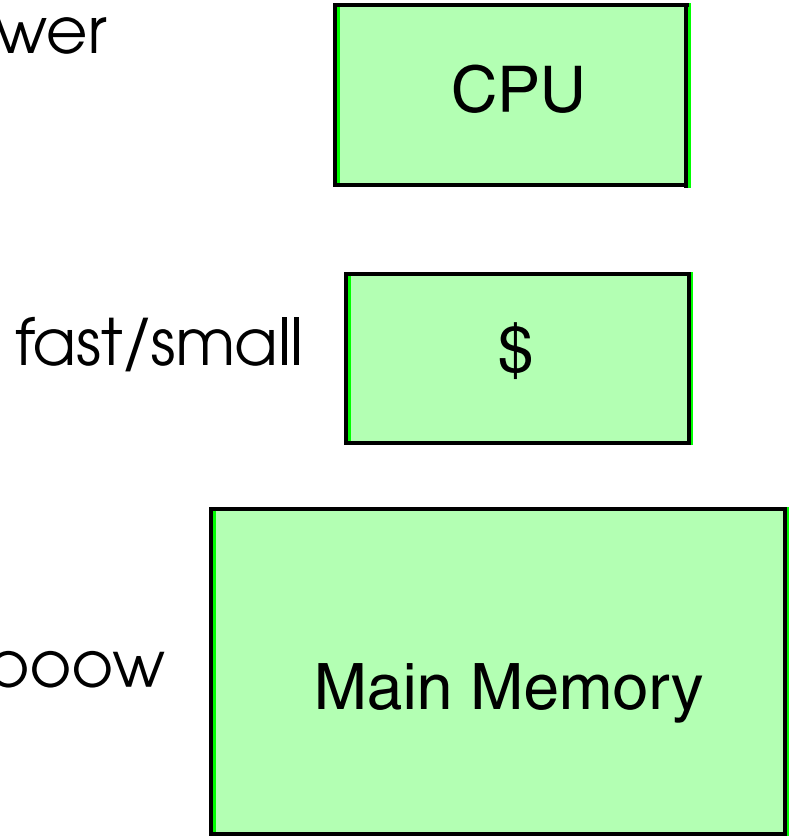
BUT CAN'T HAVE A LARGE AND FAST MEMORY

Technology

- Can tradeoff size for speed
- Can build smaller memories that are faster
- or, Large memories that are slower
- Registers for example:

  about 32-128 these days

  <1 cycle access time



Memory Hierarchy

This is terribly out of date

Mem Element	Size	Speed	Bandwidth
Register	<1K	1-5ns	8000 MB/s
L1 cache	<128K	5-10	1000
L2 cache	<4M	30-50	400
Main Memory	<4G	100	133
Disk	> 2 G	20,000,000	4

How do Programs Behave

Programs:

Recall they do not behave randomly

Locality in time (temporal locality)

if a datum is recently referenced,

it is likely to be referenced again soon

Locality in space (spacial locality)

If a datum is recently referenced,

closeby data is likely to be referenced soon

History Repeats Itself

Recall: **make common case fast**

- common: temporal and spatial locality
- fast: smaller, more expensive memory

Guess that a memory reference:

1. Will have temporal locality
2. Will have spatial locality

Or, in other words:

1. Will be accessed again
2. Others, nearby will be accessed to

Storage Presence Speculation

Key Idea: A memory location may reside in multiple places

Some are fast some are slow

Speculate: Be optimistic! What you want is in fast storage

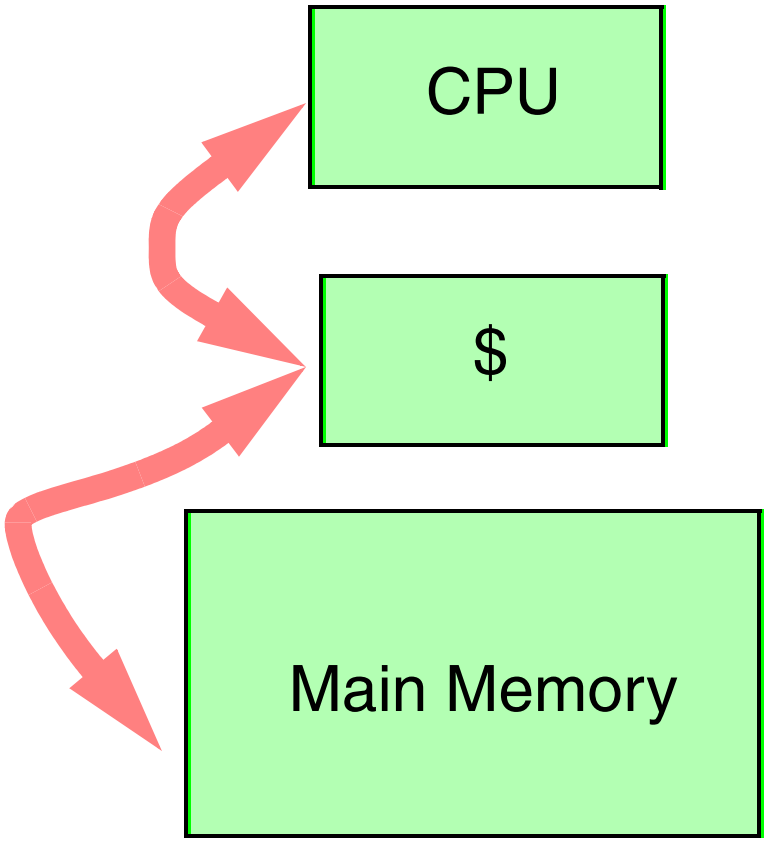
If it is good! If not, speculate it's in slow storage

then, in slower storage and so on.

This is the conventional memory hierarchy

Conventional Memory Hierarchy

- Linear: Speculate closest, on fail move down linearly



- **Correct Speculation**

  HIT

- **Mis-speculation**

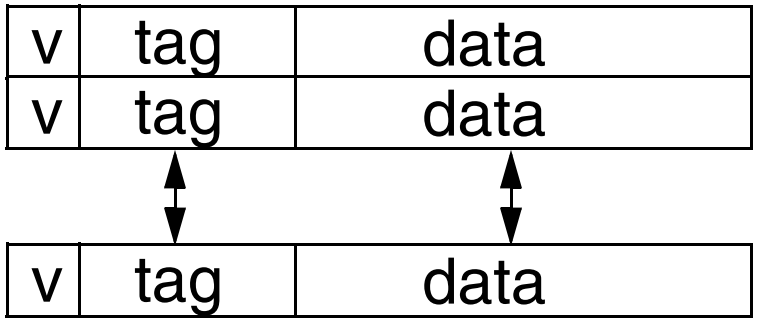
  MISS

  attempt to correct next time

  How?

  Bring in new data

Cache



put block in "block frame"

- state (e.g., valid)
- address tag
- data

block = multiple bytes, 32 very common today

Cache

on memory access

- if requested address== stored tag then
  - **HIT**
  - **return appropriate word within block**
- else
  - **MISS**
  - ***<< replace old block >>***
  - ***get block from memory***
  - ***put block in cache***
  - **return appropriate word within blockW**

Terminology

block (line, page) — minimum unit that may be present

hit — block is found in upper level

miss — not found in upper level

miss ratio — fraction of references that miss

hit time — time to access upper level

miss penalty

- time to replace block in upper level + deliver block to CPU
- access time — time to get 1st word
- transfer time — time for remaining words

What Should The TAG be?

- Full address: Let's say 2^{36} addresses

  some bits are redudant
- Assume 64k cache with 32 byte blocks (16 bits, 5 bits)
- 5 bits are the index within the block (OFFSET)
- How to select the OFFSET bits?
- Common to select the LSB for OFFSET. Why?

Memory Hierarchy Performance

time is always the ultimate measure

indirect measures can be misleading

- **miss ratio**: % of accesses that miss
- like MIPS, **miss ratio** can be misleading

average access time is better

- **$t_{avg} = t_{hit} +$ miss ratio x t_{miss}**
- e.g., $t_{hit} = 1$, miss ratio = 5% $t_{miss} = 20$
- $t_{avg} = 2$

Ultimately, **Execution time is what matters**

Fundamental Questions about Caches

where can a block be placed? **block placement**

how is a block found? **block identification**

which block is replaced on a miss? **block replacement**

what happens on a write? **write strategy** (skip for now)

what is kept? **cache type**

Block Placement

fully-associative - block goes in any frame

direct-mapped - block goes in exactly one frame

set-associative - block goes in exactly one set

Frame is a block within the cache

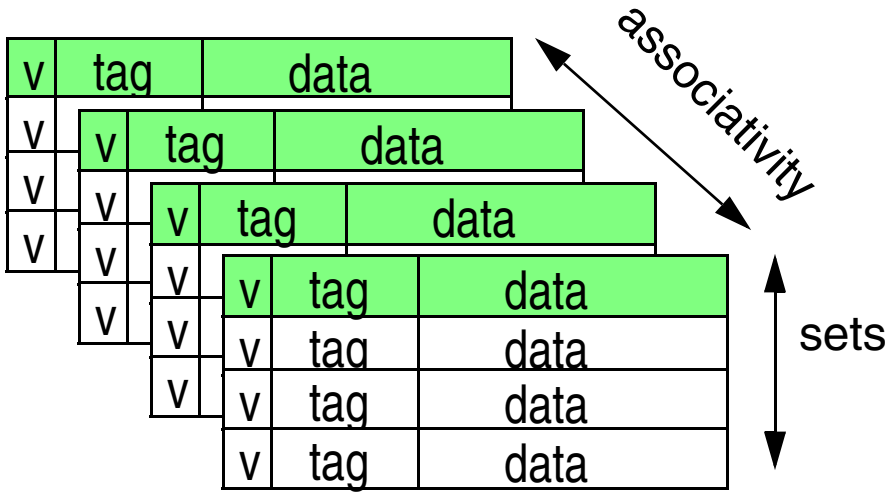
Let's look at Set-Associative Caches

Direct Mapped = Set-Associative 1

Fully-Associative = Sets == # of frames

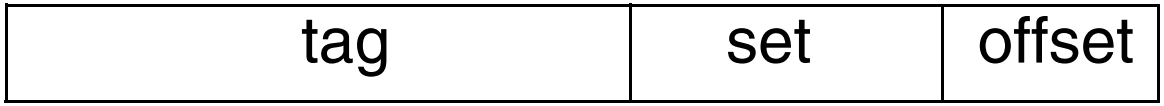
Set Associative Caches

- Locate Set
- Access all elements in the set
- Check all tags in parallel
- Select Appropriate one

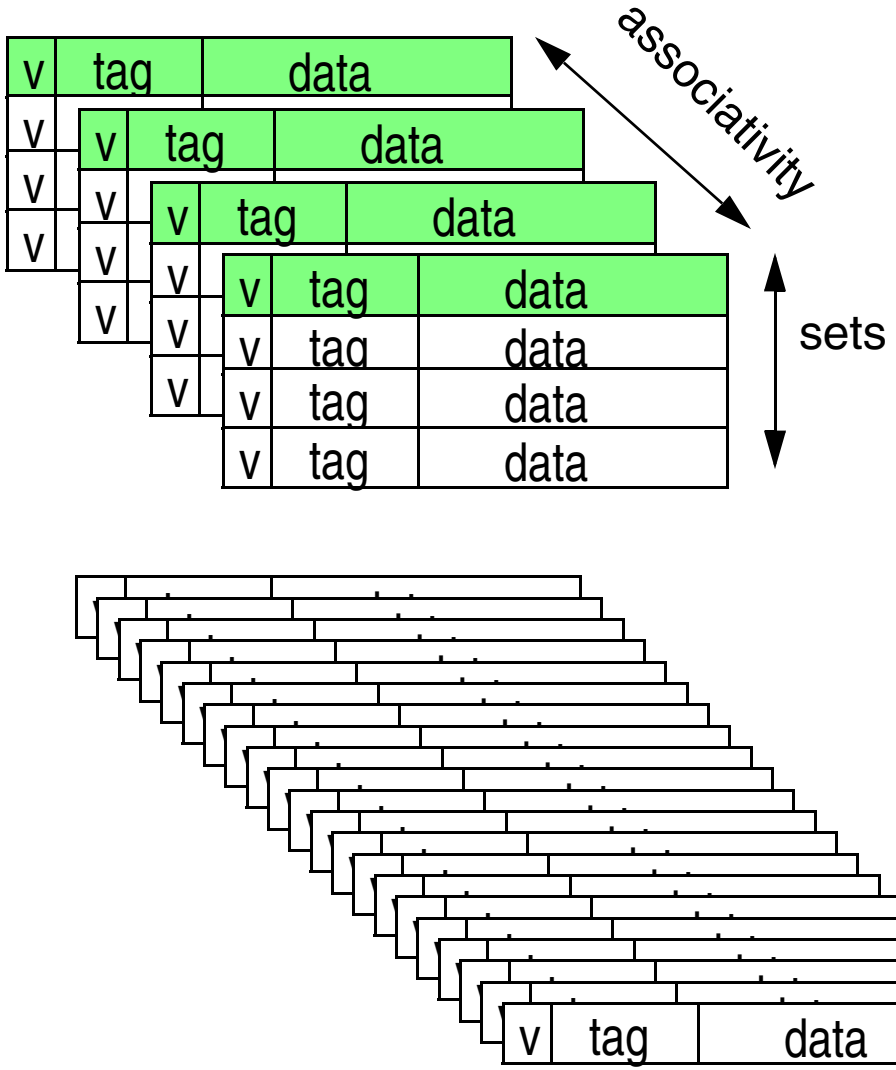


frames = associativity X sets (frames == blocks)

Size = frames X block Size



Finding and Placing a Block



v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data
v	tag	data

Block Replacement - On a Miss

least recently used - LRU

- optimized for temporal locality, complicated LRU state
- Given N blocks, how many combinations exist?

random

- pseudo-random for testing, nearly as good as LRU, simpler

not most recently used - NMRU

- track MRU, random select from others, good compromise

optimal - Belady's algorithm -

- replace block used furthest in time

Cache Handling of Data and Instructions

unified

- less costly, dynamic response, handles writes to I-stream

split I and D

- 2x bandwidth, place close to I/D ports
- can customize, poor-man's assoc, no conflicts between I/D
- self-modifying code can cause problems

caches should be split if simultaneous I and D accesses frequent

Can't just add miss rates from Split to get Unified

Interference causes different behavior

Mark Hill's Miss Classification - 3C's

compulsory — (miss in infinite cache)

- first access to a block

capacity — (miss in fullly associative cache)

- misses occur because cache not large enough

conflict

- misses occur because of mapping strategy

coherence — shared-memory multiprocessors

- misses due to invalidations from other processor (D53)

Fundamental Cache Parameters

cache size

block size

associativity

Cache Size

cache size is the total data (not including tag) capacity

- bigger can exploit **temporal locality** better
- **not ALWAYS better**

Too large a cache

- smaller is faster => bigger is slower
- access time may degrade critical path

Too small a cache

- don't exploit temporal locality well
- useful data prematurely replaced

Block Size

Block size is the data size that is both

- associated with an address tag + transfered from memory
- (advanced caches allow different)

Too small blocks

- don't exploit **spatial locality** well
- have inordinate tag overhead

Too large blocks

- useless data transfered
- useful data prematurely replaced - too few total # blocks

Associativity

Partition cache frames into

- equivalence classes (#sets) of frames each (associativity)

typical values for associativity

- 1-direct mapped, 2, 4 . . 16 - n-way associative
- Does it have to be a power of two?

larger associativity

- lower miss rate (always?), less variation among programs

smaller associativity

- lower cost, faster hit time (perhaps)

Mark Hill's "Bigger and Dumber is Better"

associativity that minimizes t_{avg} is often smaller than associativity that minimizes miss ratio

Direct-mapped vs Set associative caches with same t_{miss}

$$\textbf{diff-}t_{cache} = t_{cache}(SA) - t_{cache}(DM) >= 0$$

DM is faster than SA

$$\textbf{diff-miss} = \textbf{miss(SA) - miss(DM)} < 0$$

SA has lower MR than SA

(Actually last statement is not true in all cases, but true for most applications)

Mark Hill's "Bigger and Dumber is Better"

$t_{avg}(SA) < t_{avg}(DM)$ only if

$t_{cache}(SA) + miss(SA) \times t_{miss} < t_{cache}(DM) + miss(DM) \times t_{miss}$

$diff\text{-}t_{cache} + diff\text{-}miss \times t_{miss} < 0$

e.g.,

assuming $diff\text{-}t_{cache} = 0$ => SA better

$diff\text{-}miss = -1\%$, $t_{miss} = 20$

=> $diff\text{-}t_{cache} < 0.2$ cycle

Write Policies

Writes are harder

- reads done in parallel with tag compare; writes are not
- so, writes are slower - but does it matter?

On hits, update memory?

- yes - **write-through** (store-through)
- no - **write-back** (store-in, copy-back)

On misses, allocate cache block?

- yes - **write-allocate** (usually with write-back)
- no - **no-write-allocate** (usually with write-through)

Write-Back

- update memory only on block replacement
- dirty bits used, so clean blocks replaced w/o mem update
- traffic/reference = f_{dirty} x miss x B
- less traffic for larger caches

Write-Through

- update memory on each write
- keeps memory up-to-date
- traffic/reference = f_{writes}
- independent of cache performance

Write Buffers

buffer CPU writes

- allows reads to proceed
- stall only when full
- data dependences?
  - **detect, then stall or bypass**

Write Buffers

write policy	write alloc	hit/miss	write buffer writes to
back	yes	both	cache
back	no	hit	cache
back	no	miss	memory
thru	yes	both	both
thru	no	hit	both
thru	no	miss	memory

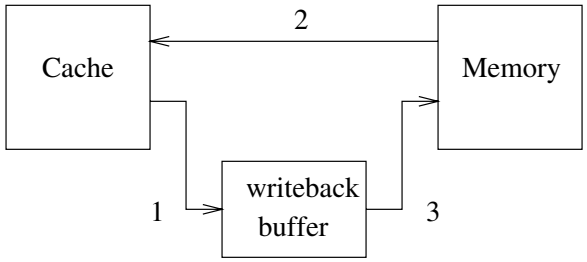
More on Write-Buffers

- design for bursts
- coalesce adjacent writes?

can also "pipeline" writes

- reads: read tag and data
- writes: read tag, save current data, write previous data

Writeback Buffers



between write-back cache and memory

- 1. move replaced, dirty blocks to buffer
- 2. read new line
- 3. move replaced data to memory

usually only need 1 or 2 writeback buffers

Advanced Caches

Caches and out-of-order scheduling/pipelines

evaluation methods

better miss rate: skewed associative caches, victim caches

reducing miss costs: column associative caches

higher bandwidth: lock-up free caches, superscalar caches

beyond simple blocks

two level caches

software restructuring

prefetching, software prefetching

Improving Latency

- Tag compare usually comes "long" after data
- Speculatively use data from cache
- Execute dependent instructions
- When tag compare completes verify speculation
- If correct good for us
- If not, need to repair
  - Specialized mechanism: Replay Buffer (Alpha)
    store dependent instructions and re-execute
  - Selectively invalidation-re-execution P4
    more on this when we talk about val pred.

Caches and Scheduling

- Think about it:

  + scheduler wakes up instructions during the cycle the results are produced

  + so that they start executing when the results are available

  + HOW? Well, if we know latencies, then simply track when each instruction will finish

- Caches are "evil":

  + Non-determinist latency

- Optimize common case:

  + Assume hit and schedule

  + replay if miss

Caches and Scheduling 2

- Early systems:

  Implicit hit/miss predictor: always hit:)

- Modern systems:

  Hit/Miss predictor (explicit)

  PC-indexed table (more a property of instruction rather than the data)

  Entry: 4-bit counter

  Increment by 4 on hit, decrement by 1 on miss (why?)

  Predict Hit if value > 8 (Or something like this)

Evaluation Methods: Hardware Counters

counts hits and misses in hardware

see Clark, TOCS 1983

+ accurate
+ realistic workloads - system, user, everything
– hard to do
– requires machine to exist
– hard to vary cache parameters
– experiments not deterministic

Evaluation Methods: Analytic Models

Mathematical expressions

+ insight - can vary parameters

+ fast

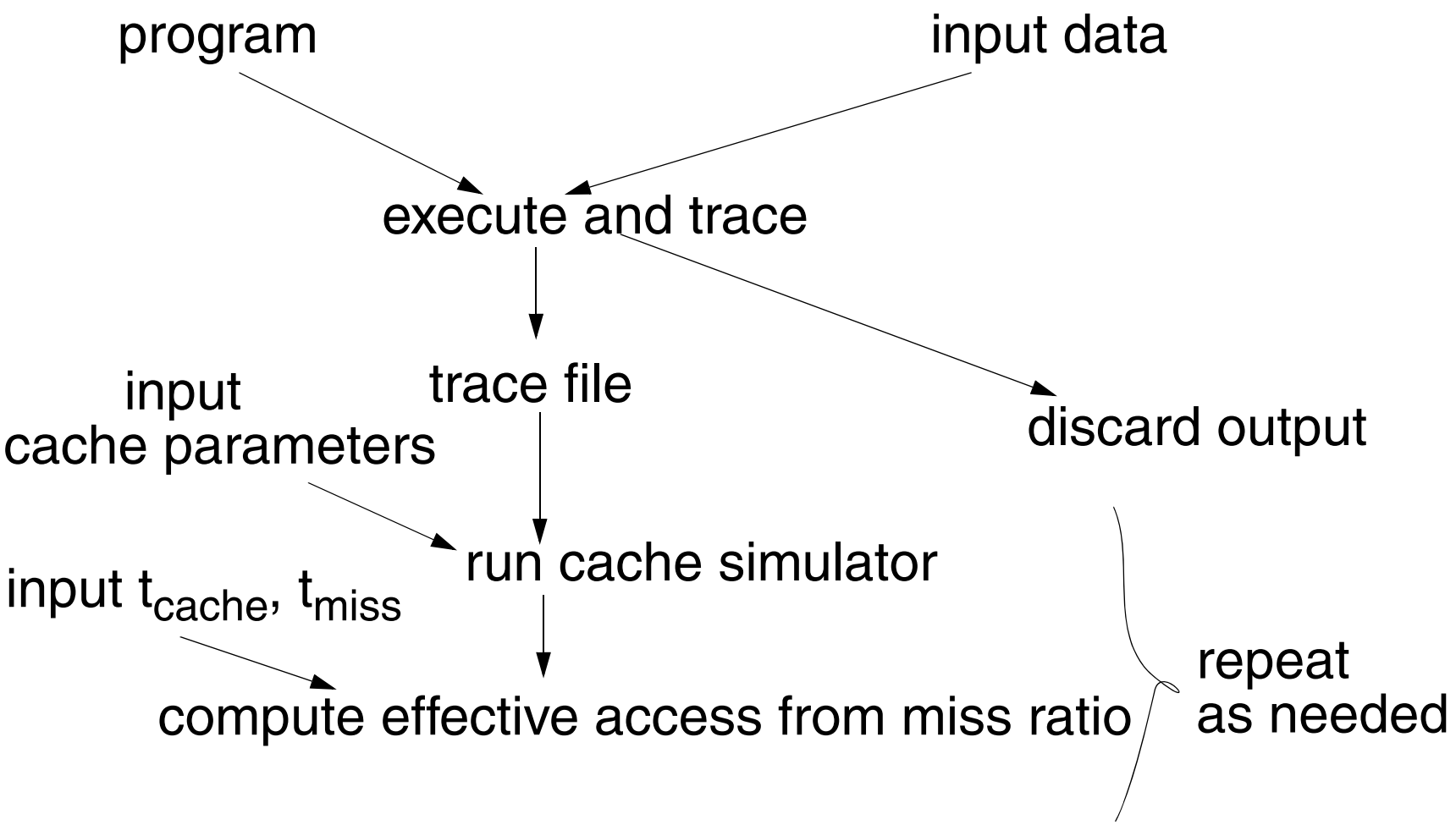
– absolute accuracy suspect for models with few parameters

– hard to determine many parameter values

Questions

- cache as a black box?
- simple and accurate?
- comprehensive or single-aspect?

Eval Methods:Trace-Driven Simulation



Eval Methods:Trace-Driven Simulation

+ experiments repeatable
+ can be accurate
+ much recent progress
– reasonable traces are very large ~ gigabytes
– simulation time consuming
– hard to say if traces representative
– don't model speculative execution

Execution-Driven Simulation

do full processor simulation each time

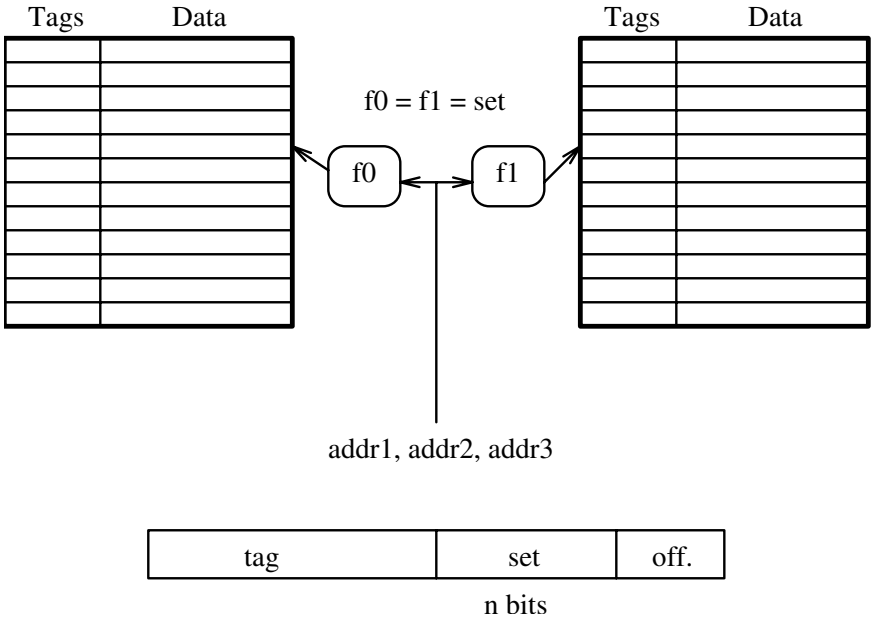
+ actual performance; with ILP miss rate means nothing
    - **non-blocking caches**
    - **prefetches (back in time?)**
    - **pollution effects due to speculation**
+ no need to store trace
– much more complicated simulation model
– time-consuming - but good programming can help

very common today

Andre Seznec's Skewed Associative Cache

conflict misses in a conventional set assoc cache

if two addresses conflict in 1 bank, they conflict in the others too



e.g., 3 addresses with same index bits will thrash in 2-way cache

Skewed Assoc. Caches

- Assume 8k 2-way set assoc with 16 byte blocks
- Conventional cache:

  blocks = $2^{13-4} = 2^9$, sets = $2^{9-1} = 2^8$ or 256

  set index = $addr_{10..4}$ (should be 8 bits)

  addresses that conflict are: 0xXXXXX**aa**X

  0x1010, 0x2010, 0x3010, repeat for ever -> always miss
- Why do they conflict?

  Because they map to the same set on every column

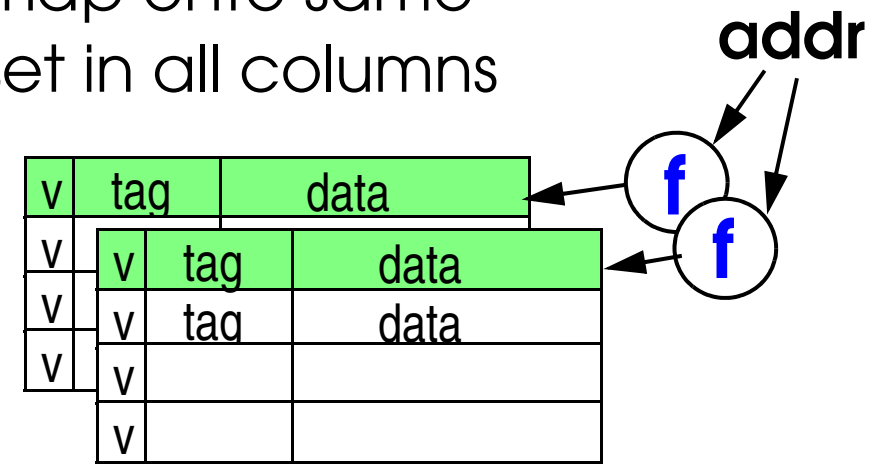
  Column? ***Physical property of cache***
- What if we use different hash functions per column?

  column 1 = 0xX..XX**aa**X column 2 = 0xX..**aa**XX

Skewed Associativity

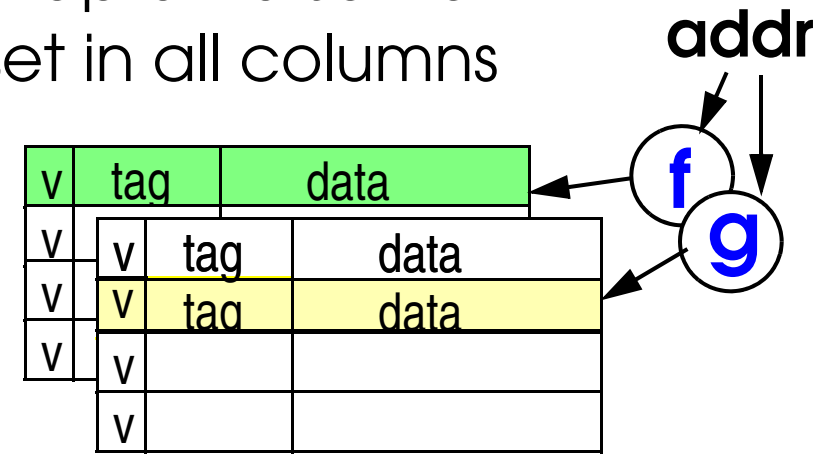
Conventional

map onto same
set in all columns



Conventional

map onto same
set in all columns



- Use different hash functions per column
- Result: Conflict in column 1 does not translate to conflict in column 2

  (actualy *may* not...)

Andre Seznec's Skewed Associative Cache

for 4-way skewed cache consider following bank functions

bank0 - a1 xor e2

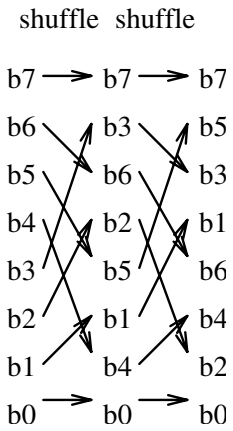
bank1 - shuffle(a1) xor a2

bank2 - shuffle(shuffle(a1)) xor a2

bank3 - shuffle(shuffle(shuffle(a1))) xor a2

Andre Seznec's Skewed Associative Cache

shuffle fucntions



implementation only adds bitwise XORs in cache access path

Column Associative Caches

- Poor man's associativity
  - High-Associativity = slow but lower miss rate (maybe)
  - Direct mapped = fast but higher miss rate
- Middle-ground
  - Organize as associative but access one column
  - if Hit = access time same as direct mapped
  - on miss, access alternate column(s)
  - slower than set associative same miss rate
  - faster than going directly to main memory
- Way Prediction (in P4)
  - Guess which column to access first

The Victim Cache

- Observation: High associativity low miss rate/high latency
- Most misses in lower associativity due to few blocks
- Exploit: **Victim Cache**

  Small cache placed in parallel with main cache

  Keeps recently evicted blocks

  Do not allocate blocks any other time
- Norm Jouppi

Victim Cache Performance

Removing conflict misses

- even one entry hepls some benchmarks
- I-cache helped more than D-cache

Versus cache size

- generally, victim cache helps more for smaller caches

Versus line size

- helps more with larger line size (why?)

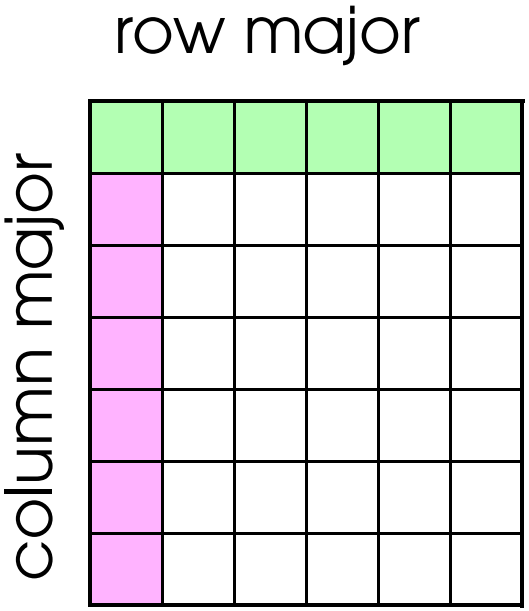
Software Restructuring

if column-major

- x(i+1, j) follows x (i,j)
- x(i,j+1) long after x(i,j)

poor code

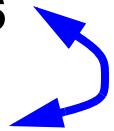
- for i = 1, rows
- for j = 1, columns
- sum = sum + x(i,j)



Software Restructuring

better code

- for j = 1, **columns**
- for i = 1, **rows**
- sum = sum + x(i,j)



optimizations - need to check if it is valid

- loop interchange (used above)
- merging arrays: physically interleave arrays
- loop fusion: two or more loops together
- blocking: operate on smaller regions at a time

Superscalar Caches

increasing issue width => wider caches

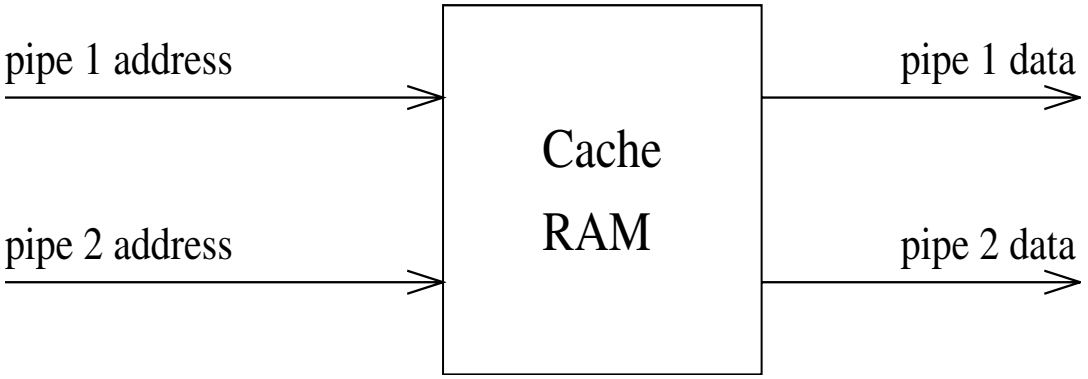
parallel cache accesses are harder than parallel functional units

- fundamental difference: caches have state, FUs don't
- operation thru one port affects future operations thru others

several approaches used

- true multi-porting
- multiple cache copies
- multi-banking (interleaving)

True Multi-porting

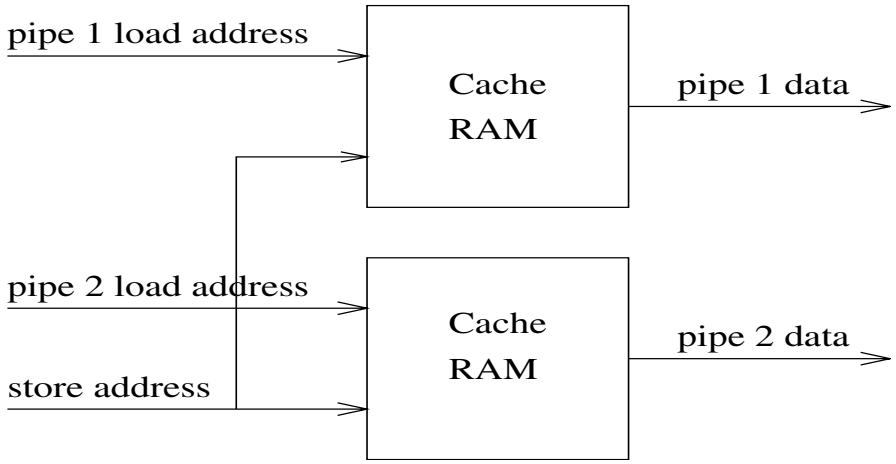


would be ideal

increases cache area

- more chip area
- slower access
- diificult to pipeline access

Multiple Cache Copies



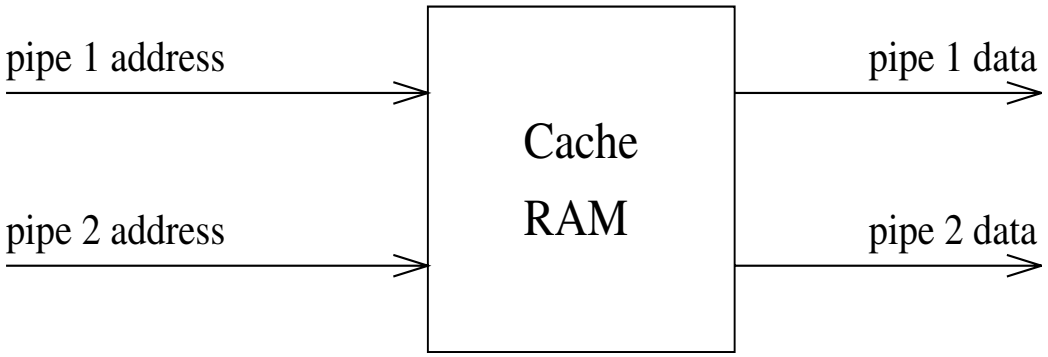
used in DEC 21164

independent load paths

single shared store path

- bottleneck, not scalable beyong 2 paths

Virtual Multi-porting



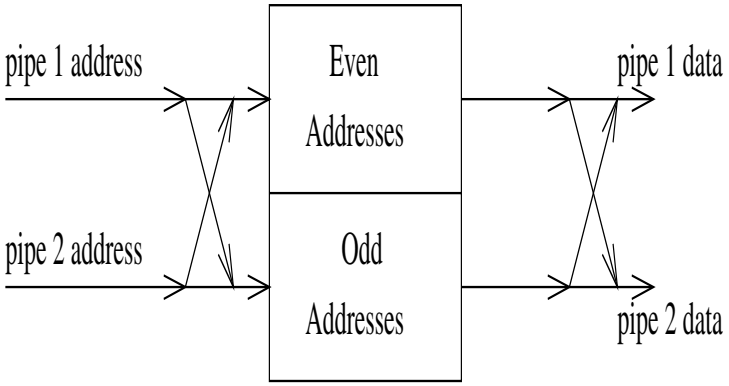
used in IBM Power2 and DEC 21264

- 21264 wave pipelining - pipeline wires WITHOUT latches

time-share a single port

- may require cache access to be faster than a clock
- probably not scalable beyond 2 ports

Multi-banking (Interleaving)



used in Intel P6(8 banks?)

need routing network

must deal with bank conflicts

extra delays can be pipelined

Beyond Simple Blocks

Break blocks into

- **address block** associated with tag
- **transfer block** to/from memory

Large address blocks

- decrease tag overhead
- but allow fewer blocks to reside
- Sector Caches (one tag per multiple blocks)
- Decoupled Sector Caches (back pointer to sector tags)

Beyond Simple Blocks

larger transfer block

- exploit spatial locality
- amortize memory latency
- but take longer to load
- replace more data already cached
- cause unnecessary traffic

Beyond Simple Blocks

address block size > transfer block size

- usually implies valid (and dirty) bit per tranfer block

was used in IBM 360/85 to reduce tag comparison logic

- 1Kbyte sectors with 64-byte subblocks

Reducing Miss Cost

if main memory takes 8 cycles before delivering 2 words/ cycle

$t_{memory} = t_{access} + B \times t_{transfer} = 8 + B \times 1/2$

B is block size in words

implies whole block is loaded before data returned to CPU

if memory returned requested word first

- cache can return it to CPU before loading it in data array
- $t_{memory} = t_{access} + MB \times t_{transfer} = 8 + 2 \times 1/2$
- MB is memory bus width in words

Reducing Miss Cost

What if processor references unloaded word in block being loaded

- need per-word valid bits
- performance penalty significant?

Why not generalize?

- handle other references that hit before all of block is back
- handle other references to other blocks that miss

called **lock-up free caches**

Latency vs Bandwidth

latency can be handled by

- hiding (or tolerating) it - out of order issue
- reducing it - caches
- parallelism helps to hide latency
    - **but increases bandwidth demand**

It's fairly "easy" to get bandwidth, latency is more tricky

Lock-up Free Caches

Normal cache stalls while a miss is pending

lock-up free caches (kroft ISCA 1981, Sohi ASPLOS 1991)

- Process other requests while miss(es) is(are) pending

potential benefits

- overlap misses with useful work and hits
- overlap misses with each other

Lock-up Free Caches

only makes sense if processor

- handles pending references correctly
- often can do useful work under a miss - dynamic scheduled
- has misses that can be overlapped

Lock-up Free Caches

key implementation problems

- handle reads to pending miss
- handle writes to pending miss
- keep multiple requests straight

Lock-up Free Caches

MSHRs - miss status holding registers

- 1. is there already a miss?
- 2. route data back to CPU
- valid bit and tag - associatively compared on each miss
- status and pointer to block frame

Lock-up Free Caches

- for every word
  - **input ID (destination register?)**
  - **send to CPU**
  - **in input buffer**
  - **in block frame**
  - **already-overwritten**

transfers from lower level to a lock-up free cache need tags

L1-L2 bus needs to be pipelined/split-transaction

asssociative MSHRs could become bottlenecks

Prefetching

even "demand fetching" prefetches other words in block

prefetching is useless

- unless a prefetch costs less than demand miss

prefetches should

- always get data before it is referenced
- never get data not used
- never prematurely replace data
- never interfere with other cache activity

Software Prefetching

use compiler to try to

- prefetch early
- prefetch accurately

prefetch into

- register (binding)
- use normal loads? faults?
- caches (non-binding) - preferred => needs ISA support

Software Prefetching

e.g.,

```
do j= 1, cols
   do  ii = 1 to rows by BLOCK
     prefetch (&(x(i,j))+BLOCK)    # prefetch one block ahead
      do i = ii to ii + BLOCK-1
        sum = sum + x(i,j)
```

Hardware Prefetching

what to prefetch

- one block spatially ahead

when to prefetch

- on every reference
  - **hard to find if block to be prefetched already in**
- on every miss
  - **better than doubling block size**
- tagged
  - **prefetch when prefetched item is referenced**

Stream Buffers

aimed at compulsory and capacity misses

prefetch into buffers, NOT into cache

- on miss start filling stream buffer with successive lines
- check both cache and stream buffer
  - **hit in stream buffer => move line into cache**
  - **miss in both => clear and refill stream buffer**

performance

- very effective for I-caches, less for D-caches

multiple buffers to capture multiple streams (better for D-caches)

Prefetching as Prediction

Stride-Based:

Common Idiom: arrays

Markov-based:

Seen a sequence of addresses then saw address'

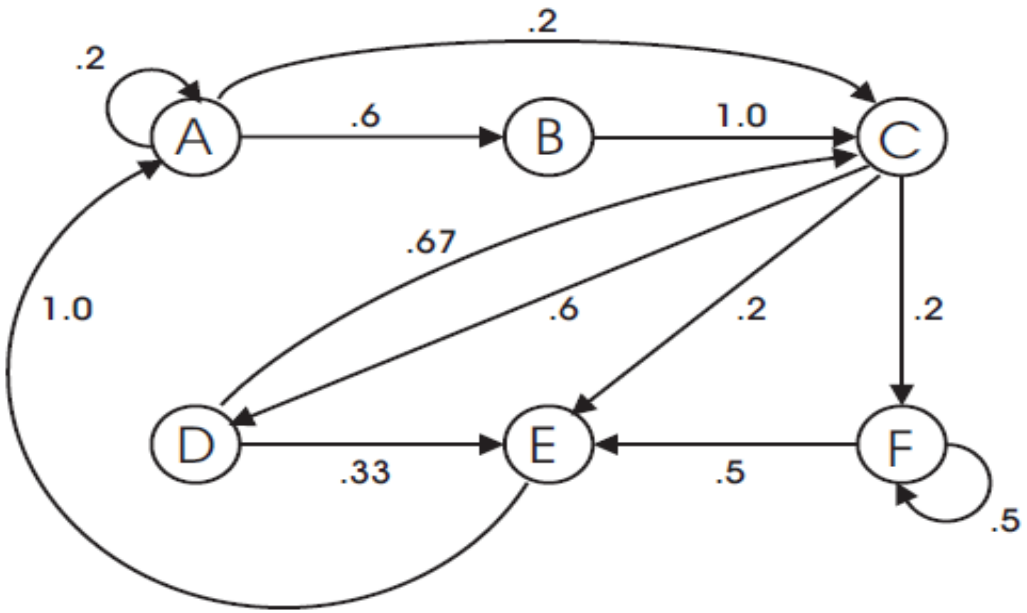
Dependence-based:

Nice for recursive data structures

Markov Prefetchers

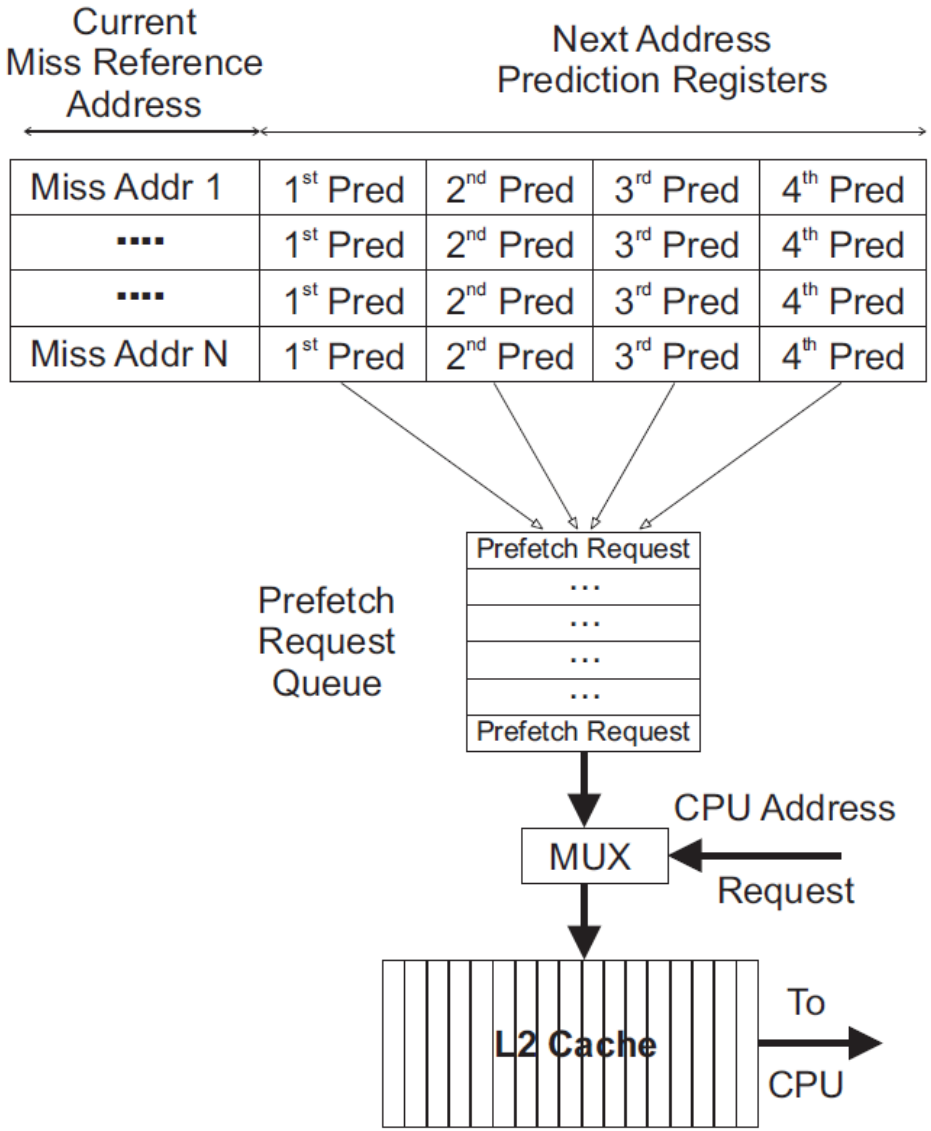
A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A, B, C, D, C

Figure 2: Sample miss address reference string. Each letter indicates a cache miss to a different memory location.



Given Address X collect info about possible next adreesses

Reference Prediction Table Prefetcher



Delta Prefetchers

compact representation

What should be the handle/Prediction?

- Use combination of PC + address
- E.g., PC xor data address
- Can predict the footprint over larger regions

Pre-computation Based Prefetching

Extract slice that leads to deliquent load

Pre-execute slice

Hardware support needed

Slice extraction can be done in software or in hardware

Why level two caches

processors getting faster w.r.t main memory

- larger caches to reduce frequency of more costly misses
- but larger caches are too slow for processor
- => reduce cost of misses with a second level cache

exploit today's technological boundary

- can't put large cache on chip (true?)
- board designer can vary cost/performance

can handle synonyms for virtual L1 caches (later)

Level Two Cache Design

what is miss ratio?

- global - L2 misses after L1 / references
- local - L2 misses after L1 / L1 misses
- solo - misses as only cache / references

NUCA

Non-Uniform Access Latency Cache Architecture

Motivation: Technologies are increasingly wire-dominated

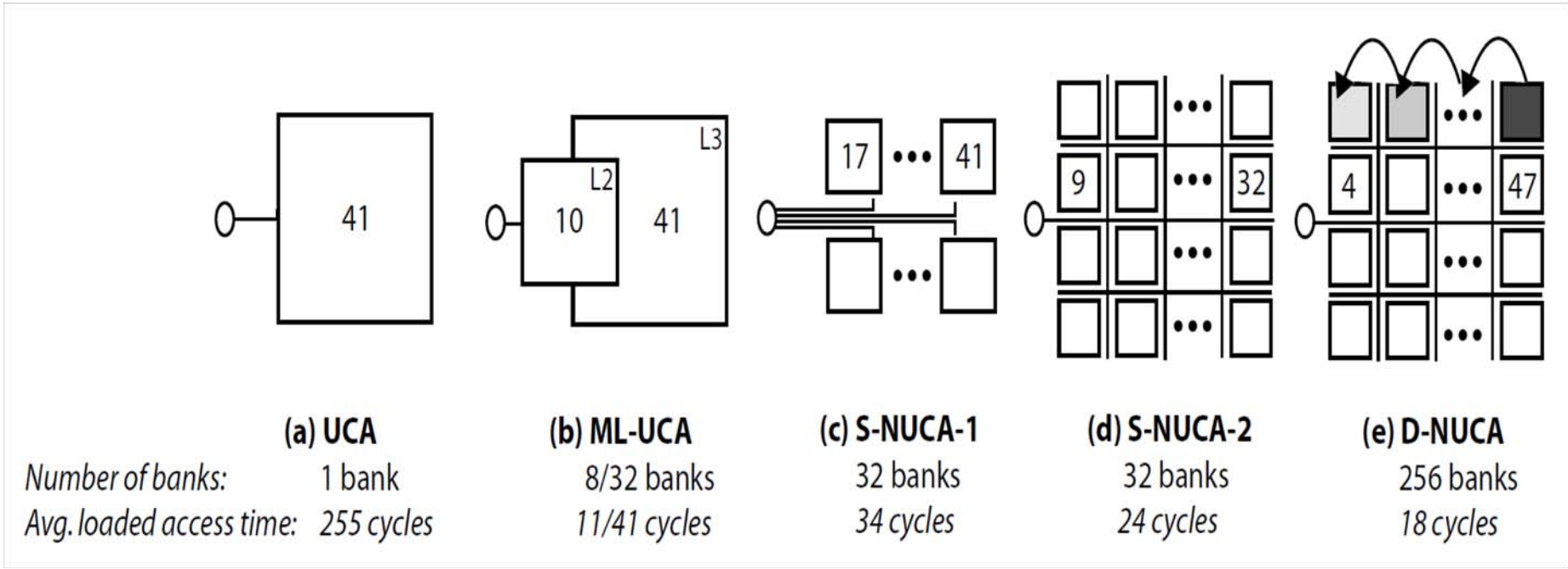
No one will build a monolithic cache

Will have a collection of cache banks

It will take variable latency to reach each of them

Opportunity: Expose latency to architecture

Cache Organization



Mapping of Sets to Banks

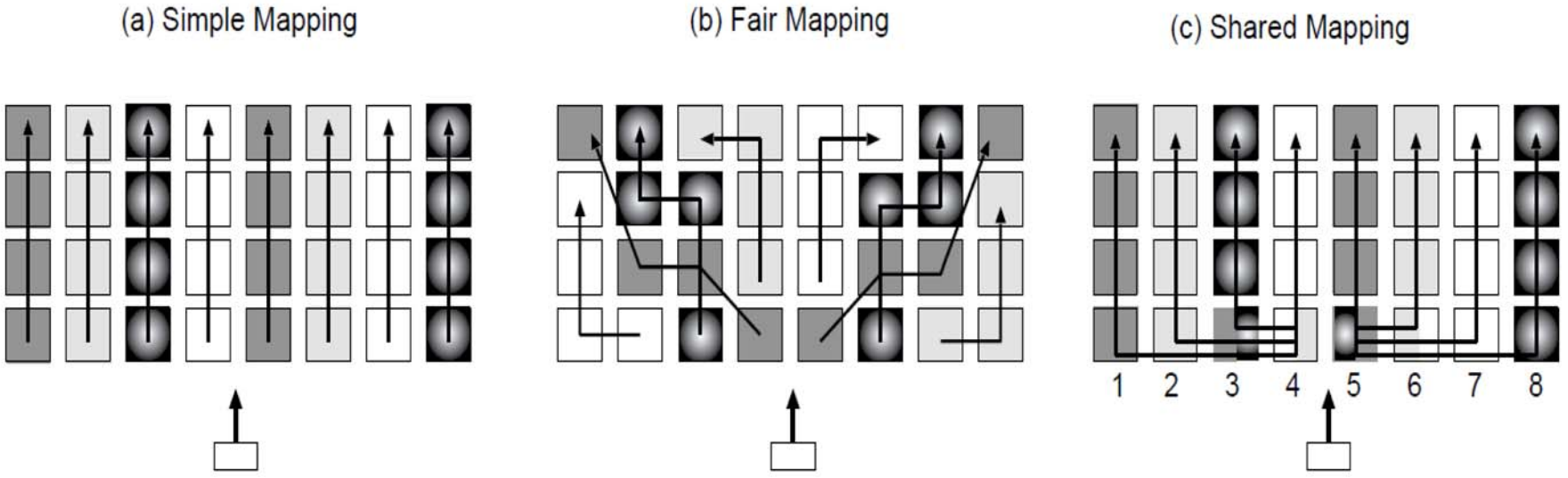


Figure 4: Mapping bank sets to banks.

Finding a Block

Incremental Search

Multicast Search

Smart Search: requires partial tags

Cache As A Prediction Mechanism

- We implicitly guess that upon referencing A
  - we will reference A again
  - we will reference A+/- delta
  - both soon
- Very good guess but not always right. Can we improve?
- Split Spatial/Temporal locality
  - Arrays vs. Non-Arrays
  - Access vector per block
- Utility counts per block (block sets really / superblocks)

Improving Cache Performance Summary

avg access time = hit time + miss rate x miss penalty

reduce miss rate

- large block size
- higher associativity
- victim caches
- skewed-associative caches
- hardware prefetching
- compiler controlled prefetching
- compiler optimizations

Improving Cache Performance Summary

reducing cache miss penalty

- give priority to read misses over writes
- subblock placement
- early restart and critical word first
- non-blocking caches
- 2nd level caches

Improving Cache Performance Summary

reducing hit time

- small and simple caches
- avoding translation during L1 indexing (later)
- pipelining writes for fast write hits
- subblock placement for fast write hits in write through cach