Chapter 5: Memory Systems

Basic caches - Appendix C
  • introduction
  • fundamental questions
  • cache size, block size, associativity

advanced caches
main memory systems

Motivation

CPU wants
  • memory reference/insn * bytes-per-reference * IPC/Cycle
  • 1.2 x 4 x 1/2ns = 2.4 GB/s

CPU can go only as fast as memory can supply

Motivation for Hierarchy

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,
  neighbouring data is likely to be referenced soon

Memory Hierarchy

make common case fast
  • common: temporal and spatial locality
  • fast: smaller, more expensive memory
Memory Hierarchy

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Latency</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>&lt; 1 KB</td>
<td>0.5 ns</td>
<td>100 GB/s</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>&lt; 256 KB</td>
<td>2 ns</td>
<td>20 GB/s</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>&lt; 256 MB</td>
<td>20 ns</td>
<td>5 GB/s</td>
</tr>
<tr>
<td>Memory</td>
<td>&lt; 64 GB</td>
<td>100 ns</td>
<td>2 GB/s</td>
</tr>
<tr>
<td>Disk</td>
<td>&gt; 512 GB</td>
<td>10 ms</td>
<td>100 MB/s</td>
</tr>
</tbody>
</table>

the numbers may change but the trends are valid

Cache

cache managed by hardware

keep recently accessed block
- temporal locality

break memory into blocks (several bytes)
- spatial locality

transfer data to/from cache in blocks

CPU

$\$\$

Main Memory
Cache on memory access
- if incoming tag == stored tag then HIT
- else MISS
  - << replace old block >>
  - get block from memory
  - put block in cache
  - return appropriate word within block

Cache Example
Memory words:
0x11c 0xe0e0e0e0
0x120 0xffffffff
0x124 0x00000001
0x128 0x00000007
0x12c 0x00000003
0x130 0xababab

Cache Example
a 16-byte cache block frame:
- state tag data
- invalid 0x?? ???

lw $4, 0x128
Is tag 0x120 in cache? (0x128 mod 16 = 0x128 & 0xffffffff)
No, get block
- state tag data
- valid 0x129 0xffffffff, 0x1, 0x7, 0x3

Cache Example
Return 0x7 to CPU to put in $4
lw $5, 0x124
Is tag 0x120 in cache?
Yes, return 0x1 to CPU
### Cache Example

Often
- cache 1 cycle
- main memory 20 cycles

Performance for data accesses with miss ratio 0.1

\[
\text{mean access} = \text{cache access} + \text{miss ratio} \times \text{main memory access} = 1 + 0.01 \times 20 = 1.2
\]

Typically caches 64K, main memory 64M
- 20 times faster
- 1/1000 capacity but contains 98% of references

---

### Eg. Cache (see 365 notes if unfamiliar)

![Diagram of a cache structure](image)

- **Address (showing bit positions)**
  - 31 30 .... 20 19 18 17 16
  - 15 14 13 12 11 10 9 8
  - Data
  - Valid Tag
  - 16 bits 32 bits
  - 16K entries

- **4KB, 4-Way Associative, 4-Byte Blocks**
  - **Index** 0 1 2
  - **Tag** 253 254 255
  - **Data**

- **4-to-1 multiplexer**
**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

**Memory Hierarchy Performance**

- time is always the ultimate measure
- indirect measures can be misleading
  - like MIPS, miss ratio can be misleading
- average access time is better
  - \( t_{\text{avg}} = t_{\text{hit}} + \text{miss ratio} \times t_{\text{miss}} \)
  - e.g., \( t_{\text{hit}} = 1, \) miss ratio = 5% \( t_{\text{miss}} = 20 \)
  - \( t_{\text{avg}} = 2 \)

**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
Block Placement

e.g. where does block 12 (1100) go?

1) Fully Associative
2) Set Associative
3) Direct Mapped

Block Identification

How to find the block

- tag comparisons
- parallel search to find lookup
- check valid bit

Block Replacement

which block to replace on a miss?

least recently used - LRU

- optimized for temporal locality, complicated LRU state
random

- pseudo-random for testing, nearly as good as LRU, simpler
not most recently used - NMIII

- track MRU, random select from others, good compromise
optimal - Belady's algorithm - replace block used furthest in time
Cache Type

**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
- true for modern pipelines

---

**Mark Hill’s Miss Classification - 3C’s**

**compulsory** — (miss in infinite cache)
- first access to a block

**capacity** — (miss in fully 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 (EE666)

---

**Fundamental Cache Parameters**

- cache size
- block size
- associativity

---

**Cache Type**

**I and D split cache**
- $I_{\text{miss}}$ is 5% and $D_{\text{miss}}$ is 6%
- 75% references are instruction fetches
- $t_{\text{avg}} = (1 + 0.05 \times 10) \times 0.75 + (1 + 0.06 \times 10) \times 0.25 = 1.5$

**Unified cache**
- $t_{\text{avg}} = 1 + 0.04 \times 10 = 1.4$ WRONG!
- $t_{\text{avg}} = 1.4 + \text{cycles-lost-to-interference}$
- will cycles-lost-to-interference be < 0.1?
- NOT for modern pipelined processors!
Cache Size

Cache size is the total data (not including tag) capacity of cache
- 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 + transferred from memory
- advanced caches allow different

Too small blocks
- don’t exploit spatial locality well
- have inordinate tag overhead

Too large blocks
- useless data transferred
- 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

Larger associativity
- lower miss rate, 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 $t_{miss}$

$\text{diff-}t_{cache} = t_{cache}(SA) - t_{cache}(DM) \geq 0$

$\text{diff-miss} = \text{miss(SA) - miss(DM)} < 0$
Mark Hill’s “Bigger and Dumber is Better”

\[ t_{\text{avg}}(\text{SA}) < t_{\text{avg}}(\text{DM}) \text{ only if} \]
\[ t_{\text{cache}}(\text{SA}) + \text{miss}(\text{SA}) \times t_{\text{miss}} < t_{\text{cache}}(\text{DM}) + \text{miss}(\text{DM}) \times t_{\text{miss}} \]
\[ \text{diff-}t_{\text{cache}} + \text{diff-miss} \times t_{\text{miss}} < 0 \]

\[ \text{e.g.,} \]
assuming \( \text{diff-}t_{\text{cache}} = 0 \Rightarrow \text{SA better} \)
\[ \text{diff-miss} = -1\%, \ t_{\text{miss}} = 20 \]
\[ \Rightarrow \text{diff-}t_{\text{cache}} < 0.2 \text{ cycle} \]

---

Mark Hill’s “Bigger and Dumber is Better”

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 Policies

### write-back
- update memory only on block replacement
- dirty bits used, so clean blocks replaced w/o mem update
- traffic/reference = \( f_{\text{dirty}} \times \text{miss} \times B \)
- traffic/reference = \( 1/2 \times 0.05 \times 4 = 0.10 \)
- less traffic for larger caches

### write-through
- update memory on each write
- keeps memory up-to-date
- traffic/reference = \( f_{\text{writes}} = 0.20 \)
- independent of cache performance

## Write Buffer (== store Q part of ld/st Q)

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

## Write Buffers

<table>
<thead>
<tr>
<th>write policy</th>
<th>write alloc</th>
<th>hit/miss</th>
<th>write buffer writes to</th>
</tr>
</thead>
<tbody>
<tr>
<td>back</td>
<td>yes</td>
<td>both</td>
<td>cache</td>
</tr>
<tr>
<td>back</td>
<td>no</td>
<td>hit</td>
<td>cache</td>
</tr>
<tr>
<td>back</td>
<td>no</td>
<td>miss</td>
<td>memory</td>
</tr>
<tr>
<td>thru</td>
<td>yes</td>
<td>both</td>
<td>both</td>
</tr>
<tr>
<td>thru</td>
<td>no</td>
<td>hit</td>
<td>both</td>
</tr>
<tr>
<td>thru</td>
<td>no</td>
<td>miss</td>
<td>memory</td>
</tr>
</tbody>
</table>
Write Buffer

more on write buffers
• design for bursts
• coalesce adjacent writes?
• four is typical number

can also “pipeline” writes
• reads: read tag and data
• writes: read tag, save current data, write previous data

Advanced Caches

evaluation methods
reduce conflict misses: skewed associative caches, victim caches
reduce capacity & compulsory misses: prefetching (s/w & h/w), stream buffers
improve general locality, reduce conflict misses: software restructuring schemes
higher bandwidth: superscalar caches, coalescing line buffer
reducing miss cost: beyond simple blocks, requested word 1st, lock-up free caches, two level caches

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

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

+ 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

Eval Methods: 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

Nature of conflict misses

Most accesses do not conflict even in a direct-mapped cache

- except in extreme pathological cases

Those that conflict, do so repeatedly

- why? locality

Only a few blocks conflict at a given time

- # conflicts is high even with a few blocks because of repeated access

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

Andre Seznec’s Skewed Associative Cache

solution: use different mapping functions into banks

for 4-way skewed cache consider following bank functions

bank0 - A1 xor A2

bank1 - shuffle(A1) xor A2

bank2 - shuffle(shuffle(A1)) xor A2

bank3 - shuffle(shuffle(shuffle(a1))) xor A2

if we use same index bits for f0 & f1, f0 will map the 3 accesses to same set and so will f1, though f0’s set is different from f1’s but no better than 2-way assoc => jumble bits so f0, f1 get different bits

A1, A2, A3, A4 are from previous slide
Andre Seznec's Skewed Associative Cache

shuffle functions (add no delay: simply jumble up wiring!)

\[
\begin{align*}
& b_7 & b_7 & b_7 \\
& b_6 & b_3 & b_5 \\
& b_5 & b_6 & b_3 \\
& b_4 & b_2 & b_1 \\
& b_3 & b_5 & b_6 \\
& b_2 & b_1 & b_4 \\
& b_1 & b_4 & b_2 \\
& b_0 & b_0 & b_0
\end{align*}
\]

implementation only adds bitwise XORs in cache access path

Skewed Associative Cache: Performance

simple experiment

512 lines

x blocks, random access

iteratively read 10 times

how many valid blocks are in cache?

• after 1 read? after 10 reads?

results

• data dispersion, self data reorganization

Skewed Associative Cache: Performance

trace driven simulation

• execution time model is lame - time = idealtime*10N_{miss}
• simulate original, blocked, blocked+copy
• relative placement of arrays varies

results

• high variable perf for DM, SA with original and blocked
• more stable for blocked+copy (s/w scheme, cover later)
• but perf can degrade for blocked+copy (s/w overhead)
• stable for skewed for all three methods

Norm Jouppi's Victim Caches

targeted at conflict misses

victim cache: a small fully associative cache

• holds victims replaced in direct-mapped or low-assoc
• LRU replacement
• a miss in cache + a hit in victim cache
  • => move line to main cache

why would it work? think of the nature of conflicts
### Norm Jouppi's Victim Caches

<table>
<thead>
<tr>
<th>tags</th>
<th>data</th>
<th>direct mapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>victim</td>
<td>hit in victim cache</td>
<td></td>
</tr>
<tr>
<td>tags</td>
<td>data</td>
<td>fully assoc.</td>
</tr>
</tbody>
</table>

### Victim Cache Performance

- Removing conflict misses
  - Even one entry helps 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?)

### Reduce Cap. & Comp. misses: Prefetching

Previously we saw reducing conflict misses

- 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

### Nature of capacity misses

Program touches lots of data - cache capacity not enough

Such large data is reused

- If no reuse, misses would be one-time and no big deal
- With reuse - by the time program comes back to a data, it is replaced from cache

So prefetch needs to bring in next data before program accesses

- Without removing currently useful data
- Prefetch needs to stay ahead of the program
Software Prefetching

use compiler to try to prefetch
- EARLY to hide latency
- ACCURATELY to avoid bandwidth wastage

prefetch into
- register (binding)
  - what if there is a later store to same address?
  - ensure no such st between prefetch & ld -- hard
- use normal loads? faults?
- caches (non-binding) - preferred => needs ISA support - if later store that store will store to cache - need not ensure

Software Prefetching

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]
Different than blocking which targets reuse -
- prefetching effective even if no (immediate) reuse
- prefetch could replace blocking but needs b/w - blocking bring once & repeatedly hits but prefetch brings repeatedly

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

Norm Jouppi’s 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)
Software Restructuring: General locality

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]

better code
  • for j = 1, columns
  • for i = 1, rows
  • sum = sum + x[i,j]

optimizations - need to check if it is valid to do them
  • loop interchange (used above)
  • merging arrays
  • loop fusion
  • blocking or tiling

Array merging: improve general locality

for (i = 1 to N)
  x = A[i] + B[i]  # A and B conflict in the cache

Merge arrays A and B into AB whose element is struct of a & b:
for (i = 1 to N)
  x = AB[i].a + AB[i].b  # a & b sequential (likely same block), no conflict

works if A and B are ALWAYS used together
  • if only one is used the other is unnecessarily in same block

Loop Fusion: improve general locality

for (i = 1 to N)  # loop 1
  x = A[i]  # assume A[] too large to fit in cache
for (i = 1 to N)  # loop 2
  y = A[j]  # later i iterations evict early A[i]'s from cache and early j iterations evict later A[i]'s so that j loop has no hits

Fuse the above loops together:
for (i = 1 to N)  # single loop
  x = A[i]
  y = A[i]  # all hits
Blocking or Tiling: improve general locality

while (some condition) {
for (i = 1 to N)
  x = A[i]
}  # A[i] is reused over and over in outer while-loop iterations

but like previous slide, later i iterations evict early A[i]'s so early A[i]'s miss in next while-loop iteration, and early i iterations evict later A[i]'s so later A[i]'s also miss

net result: all misses

---

Blocking or Tiling

Blocked or tiled code:

for (t = tile1 to tileN)      # assume A is tiled into N tiles
  while (some condition) {
    for (j = 1 to tile_size)
      x = A[t*tile_size +j]
  }

does ALL the reuse of one tile and then goes to next tile
  • net result: except for first access to a tile, all others hit

---

Blocking Algorithms: Reduce conflict miss

blocking intended to reduce misses
  • block loops so that arrays are operated on region-at-a-time
  • but there are still self-conflicts in a block
  • there can be conflicts between different arrays
  • AND
  • array sizes may be unknown at compile-time

copying block to “work space”
may be only way to effectively avoid conflicts with blocking

---

Superscalar Caches: Higher bandwidth

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
  • virtual multi-porting
  • multi-banking (interleaving)
True Multi-porting

would be ideal
increases cache area
  • more chip area
  • slower access
  • difficult to pipeline access

Multiple Cache Copies

used in DEC 21164
independent load paths
single shared store path
  • bottleneck, not scalable beyond 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
Simulation Study - Toni Juan et al.

focus on two solutions
  - true multiporting: ideal
  - multi-banking: more scalable

use dynamically scheduled processor
  - 4-issue, 64 window ----> 32-issue, 512 window

direct-mapped 32KB cache
  - processor-cache ports
  - n-banks
  - cell-port (internal)

True Multi-ported Caches

importance of bursts
  - 8-way processor
    - avg. 1.2 requests per cycle
    - 2-port => 0.85 of 8-port performance
    - 4-port => 0.97 of 8-port performance

effect of latency (0 cycle delay => 1 cycle hit)
  - dynamic scheduling (OoO issue ) helps
  - 2-port 1 cycle better than 4-port 2-cycle
  - but 2-port 2 cycle better than 1-port 1 cycle

Multi-banked Caches

configuration
  - each bank is blocking but interleaving gives non-blocking
  - addresses use simple interleaving
  - use 4 processor-to-cache ports

tradeoffs
  - large crossbar adds to latency
  - smaller cross bar increases conflicts (less b/w)
Performance of Multi-banked Caches

bank conflicts is a problem
- multi-ported vs multi-banked non-blocking

bank blocking hurts for small #banks
- multi-banked non-blocking vs multi-banked blocking
- non-blocking here is lock-up-free not multiporting (we discuss lock-up-free later so skip this for now)

routing delays also important
- consider 1 and 2 cycle delays

Nature of Bank Conflicts

many requests are to the same bank
but many are to the same block
- 0.75 of pending requests with 4 banks are to same line

Simultaneous access to the same line can be exploited to reduce bandwidth demand:
- n requests to the same line needs only one actual access - 1/n th bandwidth
- get requested line with one access and satisfy all requests
- a cool architectural alternative to brute-force VLSI multiporting

Bank Conflicts

Fraction of All Requests vs Number of Simultaneous Requests to a Bank

Number of Banks vs IPC
**Same Line Requests**

![Graph showing the relationship between Fraction of All Conflicts and Number of Distinct Lines Among Conflicting Accesses for different bank configurations.]

**Alternative Designs**

- multi-banks with multi-ports per bank
  - improves performance considerably
  - compare 4 banks 1 port with 2 banks 2 ports
- add FIFOs to banks
  - absorbs some bank busy stalls
- add coalescing to FIFOs
  - combine requests to same line so that n requests to the same line needs only one actual access - 1/n th bandwidth

**Hybrid Performance**

![Graph showing the performance of multi-banked systems with and without FIFOs.]

**Add FIFOs performance**

![Graph showing the performance improvement with the addition of FIFOs.]

---

© 2009 by Vijaykumar
Consider Area/performance

Area/performance conclusions
multi-bank with same-line optimization seems to work best
standard multi-bank caches have relatively low cost
  • but also low performance
hybrid designs perform well but relatively high cost

Reduce miss cost(1): Beyond simple blocks
Previous schemes reduced misses, now we reduce miss cost
on-chip caches: large or small blocks?
  • pins limit bandwidth
    • increases memory-cache transfer time for large blocks => smaller block is better
  • tag overhead high for small blocks => larger block better
tag & transfer have opposite requirements => decouple tag and transfer: break blocks into
  • “address block” associated with tag
  • “transfer block” to/from memory

Beyond Simple Blocks
large address blocks
  • decrease tag overhead
  • but allow fewer blocks to reside
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 transfer block

was used in IBM 360/85 to reduce tag comparison logic
  • 1Kbyte sectors with 64-byte subblocks

Reduce Miss Cost(2): Requested word 1st

if main memory takes 8 cycles before delivering 2 words/cycle
\[ t_{\text{memory}} = t_{\text{access}} + B \times t_{\text{transfer}} = 8 + B \times \frac{1}{2} \]
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
  • \( t_{\text{memory}} = t_{\text{access}} + MB \times t_{\text{transfer}} = 8 + 2 \times \frac{1}{2} \)
  • MB is memory bus width in words

Reducing Miss Cost(3): Overlap misses

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
  • ultimately limited by physics
**Latency vs Bandwidth**

bandwidth can be handled by
- wider buses, interfaces
- banking/interleaving, multiporting
- bandwidth improvement usually increases latency
- hierarchies decrease bandwidth demand to lower levels
- parallelism puts more demand on bandwidth
- if average b/w demand is not met => infinite queues
- bursts are smoothed by queues
  - if burst is much larger than average => long queue
  - eventually increases delay to unacceptable levels

**Lock-up Free Caches: Overlap misses**

normal cache stalls while a miss is pending
lock-up free caches [kroft ISCA 1981, Sohi ASPLOS 1991]
- handle hits while 1 miss is pending
- handle hits and misses while k misses pending
- aka non-blocking caches

potential benefits
- overlap misses with useful work and hits
- overlap misses with each other
- reduces miss cost AND improves cache bandwidth

**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

transfers from lower level to a lock-up free cache need tags
L1-L2 bus needs to be pipelined/split-transaction
associative MSHRs could become bottlenecks

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)
Multi-level Inclusion

multi-level inclusion holds if L2 cache always superset of L1
- handle synonyms (later)
- filter coherence traffic (EE666)
- makes L1 writes simpler
  - for both write-through and write-back

example: local LRU not sufficient
- assume L1 holds two and L2 holds three blocks
- and both use local LRU
- processor reference: 1,2,1,3,1,4
- final contents of L1: 1,4
- L1 misses: 1,2,3,4
- final contents of L2: 2,3,4 but not 1
also block sizes can be different

Multi-level Inclusion

inclusion takes effort to maintain
- make L2 cache have bits or pointers giving L1 contents
- invalidate from L1 before replacing from L2
- number of pointers per L2 block
  - is L2 blocksize/L1 blocksize

will give a paper by Wang, Baer, Levy ISCA 1989

Level Two Cache Design

L1 cache design similar to single-level cache
- when main memory "faster" wrt CPU
apply previous experience to L2 design?
L2 "global" miss ratios not significantly altered by presence of L1
- if L2 cache size >= 8 x L1 cache size
  - Przybylski et al., ISCA 1989

But L2 caches bigger than before
**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

**Level Two Cache Example**

recall adding associativity to a single-level cache helped if

\[ \text{diff-t}_{\text{cache}} + \text{diff-miss} \times t_{\text{miss}} < 0 \]

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

\[ \Rightarrow \text{diff-t}_{\text{cache}} < 0.2 \text{ cycle} \]

**Level Two Cache Example**

consider doing the same (add associativity) in an L2 cache where

\[ t_{\text{avg}} = t_{\text{cache1}} + \text{miss1} \times t_{\text{cache2}} + \text{miss2} \times t_{\text{memory}} \]

improvement only if

\[ \text{miss1} \times \text{diff-t}_{\text{cache2}} + \text{diff-miss2} \times t_{\text{memory}} < 0 \]

\[ \text{diff-t}_{\text{cache2}} < (- \text{diff-miss2/miss1}) \times t_{\text{memory}} \]

\[ \text{diff-t}_{\text{cache2}} < 0.0005/0.05 \times 100 = 1 \text{ cycle} \]

**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
- Decouple tag and transfer blocks
- Requested word first
- Lock-up free (aka non-blocking) caches
- Second level caches

Reducing hit time
- Small and simple caches
- Avoiding translation during L1 indexing (later)
- Pipelining writes for fast write hits
- Subblock placement for fast write hits in write through cache

Main Memory

Main memory
- Memory technology: static and dynamic RAM
- Basic memory concepts
- Interleaving
- Special DRAMs
- Integrated Processor/RAM

Virtual Memory
- Basics, address translation
- TLBs, cache interaction
- Protection
- Miscellaneous
DRAM

- row/column organization
- multiplexed row/column address pins
- row address strobe - RAS, column address strobe - CAS

optimized for density, not speed
- one transistor cell acts as capacitor
- data stored as charge in capacitor
- discharge on reads => destructive reads
- charge leaks over time => refresh every 2 ms
- refresh by accessing rows
- cycle time roughly twice access time
- need to precharge data lines before access

SRAM

- row/column organization
- non-multiplexed address/data pins

optimized for speed than density
- 4-6 transistors per cell
- data stored in flipflops
- static => no refresh
- greater power dissipated than DRAM
- access time = cycle time
- density/performance (1/4 density, 4-8 performance DRAM)
Simple Main Memory

consider these parameters:
  • 1 cycle to send address
  • 6 cycles to access each word
  • 1 cycle to send word back

miss penalty for a 4-word block
  • \((1 + 6 + 1) \times 4 = 32\)

how can we speed this up?

Wider Main Memory

make memory wider
  • read out all words in parallel

memory parameters
  • same as before

miss penalty for 4-word block: \(1 + 6 + 1 = 16\)

cost: wider bus, larger expansion size, error-correction is harder
  • use banks (same as caches)
  • access banks in parallel, then send data sequentially
  • miss penalty: \(1 + 6 + 4 = 11\)

Banked or Interleaved Main Memory

Break memory into \(M\) banks so word \(A\) is in \(A \mod M\) at \(A \div M\)

Interleaved Memory

\(2^a\) by \(d\)-bit word memory module

Simple Interleaving

Complex Interleaving
Interleaved Memory Examples

for each memory module A=2 (access time), A+B=4 (cycle time)
T=1 (transfer)

simple

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>bank0</th>
<th>bank1</th>
<th>bank2</th>
<th>bank3</th>
<th>steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b/t</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>b/t</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>a</td>
<td>t/a</td>
<td>a</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>a</td>
<td>a</td>
<td>t/a</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>b/t</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b/t</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Complex interleaving non-unit stride

addresses 12 15 18 stride = 3

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>bank0</th>
<th>bank1</th>
<th>bank2</th>
<th>bank3</th>
<th>steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>a</td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>a</td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>b/t</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b/t</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>a</td>
<td>a</td>
<td>b/t</td>
<td>b</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>27</td>
<td>a</td>
<td>b/t</td>
<td>b</td>
<td>a</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
<td>b/t</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>*</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>a</td>
<td>a</td>
<td>b/t</td>
<td>a</td>
<td>*</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>a</td>
<td>b/t</td>
<td>a</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

Interleaved Memory Examples

complex

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>bank0</th>
<th>bank1</th>
<th>bank2</th>
<th>bank3</th>
<th>steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>a</td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>a</td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>b/t</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>b</td>
<td>b/t</td>
<td>a</td>
<td>a</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>a</td>
<td>b/t</td>
<td>b</td>
<td>a</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b/t</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>b/t</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>*</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>b</td>
<td>b/t</td>
<td>a</td>
<td>a</td>
<td>*</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td>a</td>
<td>*</td>
</tr>
</tbody>
</table>

Complex interleaving non-unit stride

addresses 12 14 16 stride = 2

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>bank0</th>
<th>bank1</th>
<th>bank2</th>
<th>bank3</th>
<th>steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>a</td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>a</td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>a</td>
<td></td>
<td></td>
<td>b</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>a</td>
<td></td>
<td></td>
<td>b</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stall</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stall</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Interleaved Memory conclusions**

simple interleaving for sequential accesses
- load cache words
- good for writeback caches

complex otherwise

do both
- banks - simple, high bandwidth, caches
- superbanks - multiple cache accesses (nonblocking)

how many banks? simplistic answer
- $2 \times \frac{\text{words/cycle demanded}}{\text{words/cycle/bank}}$

---

**Processor Memory Bandwidth Balance**

DRAM bandwidth not kept up with processor performance
- DRAM improves 7%/year processor improves 40%/year

Eg. balancing bandwidths.
- processor bandwidth requirement
  - 4ns clock, no data cache, 1 64-bit ld/st per clock
- memory supply
  - 120ns DRAM, 16Mbit => at least 32 banks
  - 64Mb x 1 DRAM => 2048 DRAMS (4 Gbyte)
  - 4Mb x 4 DRAM => 512 DRAMS (1 Gbyte)
  - not practical

---

**Example**

let us add a cache
- 5% miss rate
- 4 words per line
- write-back 25% dirty lines
- => approx 1 words per 4 processor cycles (cut by 4)

minimum memory still large
- 16Mb x 1 DRAM => 512 DRAMs (1 Gbyte)
- 4Mb x 4 DRAM => 128DRAMs

---

**Example**

one solution: additional levels of cache
- adds SRAM cost

another solution:
- make memory wider
- OR make better use of very large DRAM on-chip b/w
Modern DRAMs

DRAM-specific organization
- read entire row
- do multiple accesses from same row
- nibble mode

![Access Time Diagram](attachment:diagram.png)

Modern DRAMs

nibble mode
page mode
- row buffer acts like SRAM
- by changing column address, bits within row are accessed

static column
- like page mode, but don’t need to strobe CAS

Processor Memory Integration

the next logical step
- FP -> on-chip, 2 level caches -> on-chip, graphics -> on-chip

bandwidth problem
- memory densities growing rapidly
- memory speeds not improving fast enough
- processor - memory technologies not compatible
  - metal layers
  - DRAM - capacitance good
  - microprocessors - capacitance bad

DRAMs are commodity parts => industry wants standards
another possibility: cached DRAMs
- put a small cache on DRAM chip => cache multiple rows
Proposal for Processor Memory

use some (e.g., 10%) DRAM area for processor
already a Hitachi chip has a simple RISC core
use simple processor - get performance from memory
integrate interconnect interfaces
target mainstream applications (not high end)

important thing is tradeoffs
• technology, cost, performance
• conventional tradeoffs need to be re-examined

Virtual Memory

Old motivation
• make small memory look large
• premit common s/w on wide product line
• avoid overlays
• use main memory as a disk cache

Current motivation
• relocation, protection, fast start-up, sharing, sparse use
• memory mapped files, networks

engineered different from CPU caches - miss penalty is $10^6$ times

Virtual Memory (see 437 notes if unfamiliar)

Blocks are called pages
• typically 4K-16K
• fixed size per system

Architecture presents programs with a simple view
• memory addressed with 32-bit addresses
• $\text{lw} \: 1, \: 0x100028 => 0x100028$ is the “virtual address”
• system maps VA to physical address (PA)
• $0x100028 => 0xF028$ (page 15, offset 28 for 4K page)

Virtual Memory

someone else and I run unrelated programs each
• $\text{lw} \: 1, \: 0x100028$
• VA must map to different PA

Thus, VA allows
• use more physical memory than system has [old reason]
• think it is the only program running in memory
• think it always starts at address 0x0
• be protected from rogue programs
• start running when most of the program is still on disk
Virtual Memory

A VA miss is called a page fault
- an exception that saves the PC
- OS gains control and initiates disk access
- OS usually runs someone else in the meantime
- interrupt when disk access is complete
- original instruction restarts

Unlike cache misses, why is OS used to handle a page fault?

Virtual Memory

blocks called pags - 4K
page placement
- through s/w
- fully associative => avoid expensive misses
page identification
- address translation - virtual to physical
- indirection through page tables gives full associativity
- translation cached in translation buffer

Address Translation

<table>
<thead>
<tr>
<th>Virtual page number</th>
<th>Page offset</th>
<th>Physical page number</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Virtual page number
Page table or disk address

Physical memory
Disk storage
Virtual Memory

page replacement
  • policy via s/w => can be complex
  • approx LRU
  • maintain “working set”

write strategy
  • write-back (page level dirty bit)

Virtual Memory Architecture

option 1: per process address space
  • each process is given an address space when created
  • gone when process dies
  • commonly used

option 2: system wide shared virtual address space
  • can persist over system lifetime
  • => “single level store”
  • requires large virtual address space
  • used in some IBM systems

System-wide VM: IBM PowerPC

32-bit effective address
16 segment registers per process
combined to 52-bit virtual address
protection via OS loading of segment registers
16M segments in system

System-wide VM: IBM PowerPC

32-bit effective address
16 segment registers per process
combined to 52-bit virtual address
protection via OS loading of segment registers
16M segments in system
Per Process VM

logical path
- two memory operations
- often two levels of page tables - PTE and data!
- too slow!

Address Translation

fast path
- translation lookaside buffer (TLB, TB)
- a cache with PTEs for data
- number of entries 8 to 1024
Page Table Size

32 bit virtual address and 4K pages => about 1M pages
But modern systems have addresses > 32 bits
most have to support relatively small pages
e.g.,
  • 48-bit virtual address
  • 4K page => 36 page number bits => 64 Gpages

there is no way the entire page table can be memory resident

Reducing Page Table option 1: Virtual

multi-level page table (virtual)
  • first-level PT in kernel virtual space
  • second-level PT in physical memory
  • not all first-level pages have to exist
    • 2nd-level points to disk if 1st-level page not exist
    • else 2nd-level points to 1st-level page in memory

hardware support
  • base registers to point to PTs
  • limit registers may be provided

Reducing Page Table Size option 1: Virtual

Think of 2nd level as “page table” for the process’s page table
Page table is accessed on a TLB miss (TLB hit => no access)
On a TLB miss, an exception is raised and OS comes in
OS uses process's base register to access 2nd level (in memory)
  • if second-level points to disk then a RECURSIVE page fault
    • first-level page not in memory, and is brought
    • actual data page may or may not exist in memory
    • yes => use 1st-level to update TLB
    • no => another fault (update 1st level + TLB)
Reducing Page Table  option 2: Physical

multi-level page table (physical)
  • all levels in physical memory

hardware support
  • base register points to root PT

  e.g., DEC Alpha

Multi-level Page Tables

Option 1 - Virtual  vs. Option 2 - Physical

upon TLB miss (you not access page table as long as TLB hits)
  • option 1 - virtual: because 1st level is in OS’s VA =>
    • page table access will require OS intervention
    • => TLB miss is handled in s/w (OS)
    • => slow but because 1st level in OS’s VA => needs
      less physical memory
  • option 2 - physical: because all levels are in physical mem
    • page table access needs no OS
    • => TLB miss handled in h/w
    • => fast but needs all levels in memory => needs
      more physical memory

Option 3: Inverted/Hashed Page Table

inverted
  • hashed virtual address points to hash table
  • hash table entry points to linked list of PTEs
  • e.g., IBM POWER1

size of hash table =
  • (phys mem size / page size) x pointer size x safety factor

size of page table =
  • (phys mem size / page size) x table entry size
Inverted/Hashed Page Table

Hash Table

Virtual page number → Hash function

• Typically XOR upper and lower bits of virtual page number

Segment ID

virt. page index

24 bits

19 bits

19-n Os

n bits

Page table index

Alternative: Hashed Page Table

Hash directly into page table

• Less time but more space (due to safety factor)

Issues

• Open or closed chaining

• Synonyms

Inverted Page Table

VPN Next

Page table entry

VPN Next

Page table entry

VPN Next

Page table entry
Option 3: hashed/inverted tables

In this option
- the page table can be in physical memory
- because it is small (unlike option 2)
- if it is in OS’s VA then same issues/tradeoffs as option 1

Page table and TLB interaction

you access page tables ONLY on TLB miss
- you do NOT have multi-level entries in TLB like page table
  - instead the page table is traversed (in h/w or s/w)
  - and the FINAL physical address is put in TLB
  - and NOT the intermediate levels

page table changes due to page replacement from physical memory
- replacement => page table change => TLB MAY be stale
- will need OS to invalidate TLB called TLB shootdown
- so unlike caches TLB is visible to OS

Other TLB interaction

TLB holds VA to PA translation and is indexed by VA
Context switches: all VAs look alike => so new process will use old process’s translations
- flush the TLB at context switch (slow)
- put PIDs (process IDs) in TLB entries (better)

Address Translation/Cache Interaction

address translation

virtual page number           page offset

VPN                          PO

TLB

PFN                          PO

page frame number           page offset
Address Translation/Cache Interaction

cache lookup tags only

Sequential TLB Access

address translation before cache lookup - SLOW!

Parallel TLB Access

address translation in parallel with cache lookup - small cache

Parallel TLB Access

address translation in parallel with cache lookup - large cache
Virtual Address Synonyms

Index taken from virtual page number - synonyms problem

Solutions to Synonyms

Limit cache size to page size times associativity
  • get index from page offset

Search all sets in parallel
  • 64K 4-way cache, 4K pages - search 4 sets (16 entries)

Restrict page placement in OS
  • guarantee that index(VA) == index(PA)

Eliminate by OS convention
  • single virtual space
  • restrictive sharing model

Virtual Address Cache

Address translation after cache miss
  • fast lookup even for large caches

Must handle
  • virtual-address synonyms
  • virtual-address space changes
  • status and protection bit changes
Virtual/Physical indexing/tagging

L1 caches
• cannot do physical indexing because TLB in critical path
• so, virtual indexing and physical tagging
  • TLB gives physical tag in parallel with L1 access
  • need to solve synonyms using above schemes

Virtual/Physical indexing/tagging
• what about virtual indexing and virtual tagging?
  • no TLB on L1 hits, access TLB only on L1 miss
  • not much speed advantage over physical tag
  • but has many problems --
  • context switch => next process also starts at VA 0
  • => cache flush unless PIDs in tags
  • but if PIDs are reused by OS (PIDs not infinite)
  • => cache flush by OS upon PID reuse

L2 caches - physically indexed physically tagged
• physical index no problem because TLB done during L1

Protection

goal:
• one process should not interfere with another

Process model
• privileged kernel
• independent user processes

Privileges vs policy
• architecture provided primitives
• OS implements policy
• problems arise when h/w implements policy

Protection Primitives

user vs kernel
• at least one privileged mode
• special case of rings
• usually implemented as mode bits

how do we switch to kernel mode?
• protected “gates”
• change mode and continue at pre-determined address

h/w to compare mode bits to access rights
• only access certain resources in kernel mode
Protection Primitives

base and bounds
  • privileged registers
  • base <= address <= bounds

segmentation
  • multiple base and bound registers
  • protection bits for each segment

page-level protection
  • protection bits in page entry table
  • cache them in TLB for speed

Caches and I/O

what happens if I/O does DMA (direct memory access)
  • to write memory addresses that are currently cached

solution 1: disallow
  • require OS to flush and make inaccessible any DMA buffer
    + simple h/w
    – complicates OS and adds overhead

solution 2: cache coherence h/w
  • h/w at cache invalidates or updates data as DMA is done
  • +/- complement of above but needed for multiprocessors ..

Cache coherence

Solutions - Chapter 4 and EE666
  • no caches, shared caches, cache only read-only or private
  • cache coherence protocol
    • snooping - caches detect
    • directory - memory knows who has what