CS184b: Computer Architecture [Single Threaded Architecture: abstractions, quantification, and optimizations]

Day13: February 20, 2000

Cache and Memory System Optimization

Last Time

- Motivation for Caching
  - fast memories small
  - large memories slow
  - need large memories
  - speed of small w/ capacity/density of large
- Temporal Locality
- Miss types: capacity, compulsory, conflict
- Associativity, replacement

Today

- DRAM (Main Memory) technology
- Spatial Locality
- Worked once, do it again…
  - multi-level caching
- split, nonblocking, victim
- prefetch
- coding/compiling for

Dynamic RAM

- Conventional, commercial, bulk DRAM
- optimized for density
  - small cell size (1T, capacitor)
  - as small capacitor as can get away with
  - large arrays
  - small signal swings, slow to detect

Dynamic RAM

- Native organization is square
  - memory columns = memory rows
  - $w=2^a$
- After about 512-1024 bit-line column
  - hard to detect bit (too slow)
  - recall charge-sharing read operation
  - more rows in column→more capacitance

Dynamic RAM

- Native banks about 1Mbit (1024×1024)
  - raw internal bw 1024b per read
  - multiplexed down further for off-chip xfer
- Once read, hold data in sense-amps
- Just multiplexing to access within row
- “column” access like static RAM
  - static column mode, page mode, …
  - “cache” RAM
    - (present cache-like interface to DRAM)
  - RAMBUS: pipeline out column data

Dynamic RAM

- Large DRAMs
  - multiple banks of roughly this size (1Mb)
  - each may have column “cache”
  - overlap long latency read access with access to separate bank
    - fetch column B1
    - fetch column B2
    - …
    - read data from B1 fetched column

Synchronous DRAM

- High-speed, synchronous I/O
- Standard DRAM-like row/column addressing
- High speed pipeline/burst read of column data
- Expose banking/paging
- row access ~ 32ns
- pipeline column reads at 8ns (shrinking)
  - 125MHz

Main Memory

- Past:
  - DRAMs only provided a few output bits
  - Wide memories by using multiple DRAM components in parallel (e.g. SIMMs)
  - Larger deeper memories with multiple DRAM components on memory bus
    - adds delay sharing bus, chip crossing to RAM
    - time to select which component
  - Memory access time slower than raw DRAM time

Main Memory

- Today:
  - wider DRAM outputs
  - fewer chips needed to provide desired capacity
    - for typical/commodity systems
  - banking within DRAM
- Tomorrow?
  - IC separation disappear?

Re-Engineering DRAM

- Can engineer DRAM for speed
  - trade density for speed
- NEC example ISSCC'99
  - 8Kb "bank"
  - 250λ^2 per bit (compare 100λ^2 per bit conv.)
  - 6.8 ns random access (9.1 ns cycle)
  - 64Mb array

NEC DRAM "bank" tradeoffs

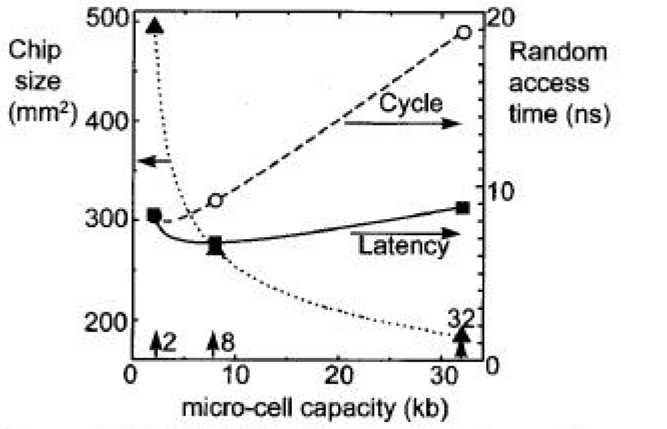


Figure 24.4.3: Relationship among micro-cell-array capacity, access times, chip size for 64Mb macro.

[ISSCC'99 p416-417]

Spatial Locality

Spatial Locality

- Higher likelihood of referencing nearby objects
  - instructions
    - sequential instructions
    - in same procedure (procedure close together)
    - in same loop (loop body contiguous)
  - data
    - other items in same aggregate
    - other fields of struct or object
    - other elements in array
    - same stack frame

Exploiting Spatial Locality

- Fetch nearby objects
- Exploit
  - high-bandwidth sequential access (DRAM)
  - wide data access (memory system)
- To bring in data around memory reference

Blocking

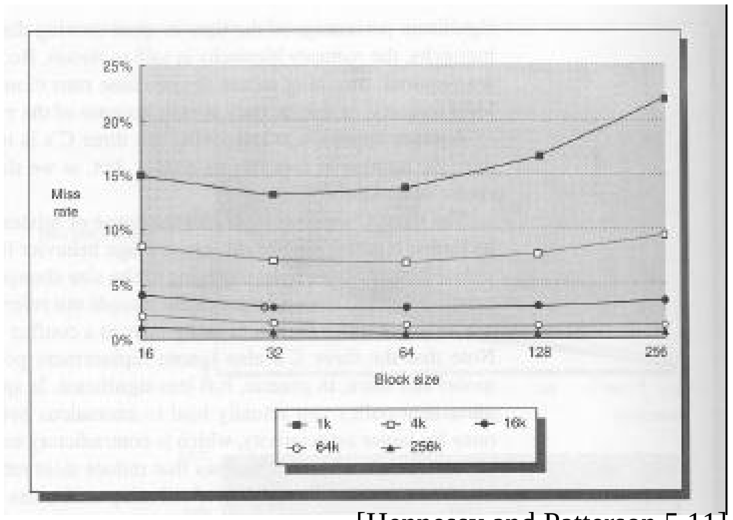
- Manifestation: Blocking / Cache lines
- Cache line bigger than single word
- Fill cache line on miss

- Size b-word cache line
  - sequential access, miss only 1 in b references

Blocking

- Benefit
  - less miss on sequential/local access
  - amortize cache tag overhead
    - (share tag across b words)
- Costs
  - more fetch bandwidth consumed (if not use)
  - more conflicts
    - (maybe between non-active words in cache line)
  - maybe added latency to target data in cache line

Block Size



[Hennessy and Patterson 5.11]

Optimizing Blocking

- Separate valid/dirty bit per word
  - don't have to load all at once
  - writeback only changed
- Critical word first
  - start fetch at missed/stalling word
  - then fill in rest of words in block
  - use valid bits deal with those not present

Multi-level Cache

Cache Numbers (from last time)

50ns main memory
1 GHz CPU

- No Cache
  - CPI=Base+0.3*50=Base+15
- Cache at CPU Cycle (10% miss)
  - CPI=Base+0.3*0.1*50=Base +1.5
- Cache at CPU Cycle (1% miss)
  - CPI=Base+0.3*0.01*50=Base +0.15

Implication (Cache Numbers)

- To get 1% miss rate?
  - 64KB-256KB cache
  - not likely to support GHz CPU rate
- More modest
  - 4KB-8KB
  - 7% miss rate
- 50x performance gap cannot really be covered by single level of cache

…do it again…

- If something works once,
  - try to do it again
- Put second (another) cache between CPU cache and main memory
  - larger than fast cache
  - hold more … less misses
  - smaller than main memory
  - faster than main memory

Multi-level Caching

- First cache: Level 1 (L1)
- Second cache: Level 2 (L2)
- CPI = Base CPI

  +Refs/Instr (L1 Miss Rate)(L2 Latency) +

  +Ref/Instr (L2 Miss Rate)(Memory Latency)

Multi-Level Numbers

- L1, 1ns, 4KB, 10% miss
- L2, 5ns, 128KB, 1% miss
- Main, 50ns

- L1 only CPI=Base+0.3*0.1*50=Base +1.5
- L2 only CPI=Base+0.3*(0.99*4+0.01*50) =Base+1.7
- L1/L2=Base+(0.3*0.1*5 + 0.01*50) =Base+0.65

Numbers

- Maybe could use L3?
  - Hypothesize: L3, 10ns, 1MB, 0.2%

- L1/L2/L3=Base+(0.3*0.1*5 + 0.01*10+0.002*50) =Base+0.15+0.1+0.1 =Base+0.35

Rate Note

- Previous slides:
  - "L2 miss rate" = miss of L2
    - all access; not just ones which miss L1
  - If talk about miss rate wrt only L2 accesses
    - higher since filter out locality from L1
- H&P: **global miss rate**
- **Local miss rate**: misses from accesses seen in L2
- Global miss rate
  - L1 miss rate $\times$ L2 local miss rate

Segregation

I-Cache/D-Cache

- Processor needs one (or several) instruction words per cycle
- In addition to the data accesses
  – Instr/Ref*Instr Issue
- Increase bandwidth with separate memory blocks (caches)

I-Cache/D-Cache

- Also different behavior
  – more locality in I-cache
  – afford less associativity in I-cache?
  – Make I-cache wide for multi-instruction fetch
  – no writes to I-cache
- Moderately easy to have multiple memories
  – know which data where

By Levels?

- L1
  - – need bandwidth
  - – typically split (contemporary)
- L2
  - – hopefully bandwidth reduced by L1
  - – typically unified

…Other Optimizations

How disruptive is a Miss?

- With
  - multiple issue
  - a reference every 3-4 instructions
- memory references 1+ times per cycle
- Miss means multiple (4,8,50?) cycles to service
- Each miss could holds up 10's to 100's of instructions...

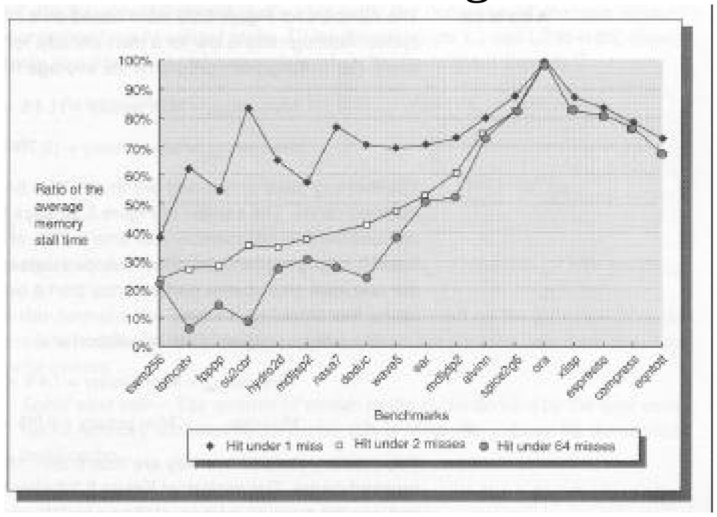
Minimizing Miss Disruption

- Opportunity:
  - out-of-order execution
    - maybe we can go on without it
    - scoreboarding/tomasulo do dataflow on arrival
    - go ahead and issue other memory operations
  - next ref might be in L1 cache
    - …while miss referencing L2, L3, etc.
  - next ref might be in a different bank
    - can access (start access) while waiting for bank latency

Non-Blocking Memory System

- Allow multiple, outstanding memory references
- Need split-phase memory operations
  - separate request data
  - from data reply (read -- complete for write)
- Reads:
  - easy, use scoreboarding, etc.
- Writes:
  - need write buffer, bypass...

Non-Blocking



[Hennessy and Patterson 5.11]

Victim Cache

- Problem with Direct-mapped (low-associativity):
  - commonly referenced data items may map to same cache location
  - force thrashing
- Mitigate:
  - add a small associative cache (buffer) to hold recent evictions

Victim Cache

- Victim Cache after L1 cache
  - not add to cycle time like assoc. check
  - like L1.5 cache :-)
  - small number of entries
- For small (4KB?) direct mapped caches
  - gives hit-rate performance of set-associative
  - …but faster (on hit case)

Prefetch

- Reduce misses, by trying to load values before they're needed
- Hardware/dynamic:
  - block cache lines an example
  - auto prefetch **next** cache block
    - to stream buffer so not pollute cache
    - should never miss on sequential control flow

Prefetch

- Software/Compiler assisted
  - exposes to model
  - may generate more memory traffic
  - requires issue slots
  - compiler can hoist/schedule access in advance of use
  - place in dominator position
    - one fetch, many uses
  - deal with cases not predictable with simple hardware heuristics
  - saw examples in VLIW/EPIC

Prefetch

- To cache
- Non-binding
- Non-faulting
- Only affects performance
  – not behavior/semantics

Coding and Compiling...

Exploit Freedom

- Much freedom exists in how we code, transform, map programs
- Can exploit that freedom
  - enhance locality
    - temporal
    - spatial
  - reduce conflicts
    - direct mapped / low associativity

Simple Thing First...

- Keep data structures small/minimal
  - at least, heavily accessed data structures

Freedom

- Data layout
  - place data referenced together close together
    - same page
    - same cache line
  - common case code together
    - bin to cache line by usage
    - even if structure large, commonly accessed data in minimum number of cache lines

Freedom

- Task sequentialization
  - process local regions of data close together in time
    - e.g. blocking, strided data access

Freedom

- Code layout
  - pack together common case (main trace)
    - close together
    - packed appropriately into cache lines
    - on same page
    - off trace code may go further away
  - make sure addresses in common traces **not** alias to same cache slot
    - compiler use feedback from program run

Implementation Specific?

- These recommendations/opt. specific to a particular microarchitecture?
  - Locality concept fairly universal
    - for current technology…
  - Optimizing locality probably good
  - Many effects depend on constants/boundaries
    - cache line size
    - blocking and size of cache at each level
    - conflicts and associativity

Big Ideas

- Structure
  - spatial locality
- Engineering
  - worked once, try it again…until won’t work
- Exploit freedom which exists in application
  - to favor what can do efficiently/cheaply