Chapter 2: Memory Hierarchy Design – Part 2

Introduction (Section 2.1, Appendix B)

Caches
  Review of basics (Section 2.1, Appendix B)
  Advanced methods (Section 2.3)

Main Memory

Virtual Memory
Fundamental Cache Parameters

Cache Size
   How large should the cache be?

Block Size
   What is the smallest unit represented in the cache?

Associativity
   How many entries must be searched for a given address?
Cache Size

Cache size is the total capacity of the cache

Bigger caches exploit temporal locality better than smaller caches

But are *not always* better

Why?
Cache Size**

Cache size is the total capacity of the cache

Bigger caches exploit temporal locality better than smaller caches

But are *not always* better

Too large a cache size

Smaller means faster $\Rightarrow$ bigger means slower

Access time may degrade critical path

Too small a cache size

Don't exploit temporal locality well

Useful data is prematurely replaced
Block Size

Block (line) size is the data size that is both
(a) associated with an address tag, and
(b) transferred to/from memory
Advanced caches allow different (a) & (b)

Problem with too small blocks

Problem with large blocks
Block (line) size is the data size that is both
(a) associated with an address tag, and
(b) transferred to/from memory
Advanced caches allow different (a) & (b)

Too small blocks
- Don't exploit spatial locality well
- Don't amortize memory access time well
- Have inordinate address tag overhead

Too large blocks cause
Block Size

Block (line) size is the data size that is both
(a) associated with an address tag, and
(b) transferred to/from memory
Advanced caches allow different (a) & (b)

Too small blocks
Don't exploit spatial locality well
Don't amortize memory access time well
Have inordinate address tag overhead

Too large blocks cause
Unused data to be transferred
Useful data to be prematurely replaced
Set Associativity

Partition cache block frames & memory blocks in equivalence classes (usually w/ bit selection)

Number of sets, $s$, is the number of classes

Associativity (set size), $n$, is the number of block frames per class

Number of block frames in the cache is $s \times n$

Cache Lookup (assuming read hit)

Select set

Associatively compare stored tags to incoming tag

Route data to processor
Typical values for associativity

1 -- direct-mapped

n = 2, 4, 8, 16 -- n-way set-associative

All blocks – fully-associative

Larger associativity

Smaller associativity
Typical values for associativity

1 -- direct-mapped
n = 2, 4, 8, 16 -- n-way set-associative
All blocks – fully-associative

Larger associativities
Lower miss ratios
Less variance

Smaller associativities
Lower cost
Faster access (hit) time (perhaps)
Advanced Cache Design (Section 2.3)

Evaluation Methods
Two Levels of Cache
Getting Benefits of Associativity without Penalizing Hit Time
Reducing Miss Cost to Processor
Lockup-Free Caches
Beyond Simple Blocks
Prefetching
Pipelining and Banking for Higher Bandwidth
Software Restructuring
Handling Writes
Evaluation Methods
Evaluation Methods

Hardware Counters

Analytic Models

Simulation
Method 1: Hardware Counters

Advantages
  +
  +

Disadvantages
  -
**Method 1: Hardware Counters**

Count hits & misses in hardware

Advantages
  +
  +
  +

Disadvantages
  -
  -

Many recent processors have hardware counters
Method 1: Hardware Counters

Count hits & misses in hardware

Advantages
  + Accurate
  + Realistic workload

Disadvantages
  - Requires machine to exist
  - Hard to vary cache parameters
  - Experiments not deterministic

Many recent processors have hardware counters
Method 2: Analytic Models

Mathematical expressions

Advantages
+ 
+ 
+ 

Disadvantages
- 
- 
- 
Method 2: Analytic Models

Mathematical expressions

Advantages

+ Insight -- can vary parameters
+ Fast

Disadvantages

- (Absolute) accuracy?
- Hard/time-consuming to determine many parameter values
Method 3: Simulation

Software model of the system driven by model of program

Can be at different levels of abstraction

   Functional vs. timing
   Trace-driven vs. execution-driven

Advantages

Disadvantages
Method 3: Simulation**

Software model of the system driven by model of program
  Can be at different levels of abstraction
    Functional vs. timing
    Trace-driven vs. execution-driven

Advantages
+ Experiments repeatable
+ Can be accurate for many detailed metrics

Disadvantages
- Slow
- Still a model, can be inaccurate
- Only provides insight for the input programs simulated
Trace-Driven Simulation

Step 1:

**Execute and Trace**

Program + Input Data $\rightarrow$ Trace File

Trace files may have only memory references or all instructions

Step 2:

Trace File + Input Cache Parameters

Run simulator

Get miss ratio, tavg, execution time, etc.

Repeat Step 2 as often as desired
Trace-Driven Simulation: Limitation?
Trace-Driven Simulation: Limitation?**

Impact of aggressive processor?
- Does not model mispredicted paths, actual order of accesses
- Reasonable traces are very large
  - (Can integrate step 1 and step 2)

Alternative: Execution-driven simulation
- Simulate entire application execution
- Model full impact of processor
  - Mispredicted paths, reordering of accesses
Average Memory Access Time and Performance
**Average Memory Access Time and Performance**

Old: Avg memory time = Hit time + Miss rate * Miss penalty
But what is miss penalty in an out of order processor?
Hit time is no longer one cycle, hits can stall the processor
Average Memory Access Time and Performance

Old: Avg memory time = Hit time + Miss rate * Miss penalty

But what is miss penalty in an out of order processor?

Hit time is no longer one cycle, hits can stall the processor

Execution cycles = Busy cycles + Cycles due to CPU stalls + Cycles due to memory stalls

Cycles from memory stalls = stalls from misses + stalls from hits

Stalls from misses = # misses * (Total miss latency – overlapped latency)
Stalls from misses = # misses * 

(Total miss latency – overlapped latency)

What is a stall?

Processor is stalled if it does not retire at its full rate
Charge stall to first instruction that cannot retire

Where do you start measuring latency?

From time instruction is queued in instruction window, or when address is generated, or when sent to memory system?

Anything works as long as is consistent

Miss latency is also made of latency due to and w/o contention

Hit stalls are analogous
What About Non-Performance Metrics?

Area, power, detailed timing

CACTI for caches

McPAT: microarchitecture model for full multicore
Figure 2.8 Relative access times generally increase as cache size and associativity are increased. These data come from the CACTI model 6.5 by Tarjan et al. (2005). The data assume typical embedded SRAM technology, a single bank, and 64-byte blocks. The assumptions about cache layout and the complex trade-offs between interconnect delays (that depend on the size of a cache block being accessed) and the cost of tag checks and multiplexing lead to results that are occasionally surprising, such as the lower access time for a 64 KiB with two-way set associativity versus direct mapping. Similarly, the results with eight-way set associativity generate unusual behavior as cache size is increased. Because such observations are highly dependent on technology and detailed design assumptions, tools such as CACTI serve to reduce the search space. These results are relative; nonetheless, they are likely to shift as we move to more recent and denser semiconductor technologies.
Figure 2.9 Energy consumption per read increases as cache size and associativity are increased. As in the previous figure, CACTI is used for the modeling with the same technology parameters. The large penalty for eight-way set associative caches is due to the cost of reading out eight tags and the corresponding data in parallel.
Multilevel Caches

Processor

L1 Inst

L1 Data

L2

Main memory
Why Multilevel Caches?
Why Multilevel Caches?**

Processors getting faster w.r.t. main memory

  Want larger caches to reduce frequency of more costly misses

But larger caches are too slow for processor

Solution: reduce the cost of misses with a second (and third!) level of cache instead
Multilevel Inclusion

Multilevel inclusion holds if L2 cache always contains superset of data in L1 cache(s)
  - Filter coherence traffic
  - Makes L1 writes simpler

Example: Local LRU not sufficient
  - Assume that L1 and L2 hold two and three blocks and both use local LRU
  - Processor references: 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
Multilevel inclusion takes effort to maintain

(Typically L1/L2 cache line sizes are different)
Make L2 cache have bits or pointers giving L1 contents
Invalidate from L1 before replacing block from L2
Number of pointers per L2 block is (L2 blocksize / L1 blocksize)
What if the L2 cache is only slightly larger than L1?
Multilevel exclusion \implies A line in L1 is never in L2 (AMD Athlon)
Level Two Cache Design

L1 cache design similar to single-level cache design when main memories were ``faster''

Apply previous experience to L2 cache design?

What is ``miss ratio''?

- Global -- L2 misses after L1 / references
- Local -- L2 misses after L1 / L1 misses

BUT: L2 caches bigger than L1 experience (several MB)

BUT: L2 affects miss penalty, L1 affects clock rate
Benefits of Associativity W/O Paying Hit Time

Victim Caches
Pseudo-Associative Caches
Way Prediction
Victim Cache

Add a small fully associative cache next to main cache

On a miss in main cache
Victim Cache**

Add a small fully associative cache next to main cache

On a miss in main cache
- Search in victim cache
- Put any replaced data in victim cache
Pseudo-Associative Cache

To determine where block is placed

Check one block frame as in direct mapped cache, but

If miss, check another block frame

E.g., frame with inverted MSB of index bit

Called a pseudo-set

Hit in first frame is fast

Placement of data

Put most often referenced data in “first” block frame and the other in the “second” frame of pseudo-set
Way Prediction

Keep extra bits in cache to predict the “way” of the next access
Access predicted way first
If miss, access other ways like in set associative caches
Fast hit when prediction is correct
Reducing Miss Cost

If main memory takes $M$ cycles before delivering two words per cycle, we previously assumed

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

where $B$ is block size in words

How can we do better?
\[ t_{memory} = t_{access} + B \times t_{transfer} = M + B \times 1/2 \]

⇒ the whole block is loaded before data returned

If main memory returned the reference first (requested-word-first) and the cache returned it to the processor before loading it into the cache data array (fetch-bypass, early restart),

\[ t_{memory} = t_{access} + W \times t_{transfer} = M + W \times 1/2 \]

where \( W \) is memory bus width in words

BUT ...
Reducing Miss Cost, cont.

What if processor references unloaded word in block being loaded?

Why not generalize?
   Handle other references that hit before any part of block is back?
   Handle other references to other blocks that miss?

Called ``lockupfree'' or ``nonblocking'' cache
Reducing Miss Cost, cont.**

What if processor references unloaded word in block being loaded?
   Must add (equivalent to) ``word'' valid bits

Why not generalize?
   Handle other references that hit before any part of block is back?
   Handle other references to other blocks that miss?

Called ``lockup-free'' or ``non-blocking'' cache
Lockup-Free Caches

Normal cache stalls while a miss is pending

Lockup-Free Caches
  (a) Handle hits while first miss is pending
  (b) Handle hits & misses until K misses are pending

Potential benefit
  (a) Overlap misses with useful work & hits
  (b) Also overlap misses with each other

Only makes sense if
Lockup-Free Caches**

Normal cache stalls while a miss is pending

Lockup-Free Caches
   (a) Handle hits while first miss is pending
   (b) Handle hits & misses until K misses are pending

Potential benefit
   (a) Overlap misses with useful work & hits
   (b) Also overlap misses with each other

Only makes sense if processor
   Handles pending references correctly
   Often can do useful work with references pending (Tomasulo, etc.)
   Has misses that can be overlapped (for (b))
Key implementation problems

(1) Handling reads to pending miss
(2) Handling writes to pending miss
(3) Keep multiple requests straight

MSHRs -- miss status holding registers

What state do we need in MSHR?
Lockup-Free Caches, cont. **

Key implementation problems

(1) Handling reads to pending miss
(2) Handling writes to pending miss
(3) Keep multiple requests straight

MSHRs -- miss status holding registers

For every MSHR – Valid bit

- Block request address
- For every word – Destination register?
  - Valid bit
  - Format (byte load, etc.)
- Comparator (for later miss requests)

Limitation?
Lockup-Free Caches, cont.**

Key implementation problems

(1) Handling reads to pending miss
(2) Handling writes to pending miss
(3) Keep multiple requests straight

MSHRs -- miss status holding registers

For every MSHR – Valid bit

   Block request address

For every word – Destination register?

   Valid bit

   Format (byte load, etc.)

   Comparator (for later miss requests)

Limitation: Must block on next access to same word
Beyond Simple Blocks

Break block size into
  Address block associated with tag
  Transfer block transferred to/from memory

Larger address blocks
  Decrease address tag overhead
  But allow fewer blocks to be resident

Larger transfer blocks
  Exploit spatial locality
  Amortize memory latency
  But take longer to load
  But replace more data already cached
  But cause unnecessary traffic
Address block size > transfer block size
   Usually implies valid (& dirty) bit per transfer block
   Used in 360/85 to reduce tag comparison logic
      1K byte sectors with 64 byte subblocks

Transfer block size > address block size
   ``Prefetch on miss''
   E.g., early MIPS R2000 board
Prefetching

Prefetch instructions/data before processor requests them

Even "demand fetching" prefetches other words in the referenced block

Prefetching is useless unless a prefetch "costs" less than demand miss

Prefetches should

???
Prefetching

Prefetch instructions/data before processor requests them

Even "demand fetching" prefetches other words in the referenced block

Prefetching is useless unless a prefetch "costs" less than demand miss

Prefetches should

(a) Always get data back before it is referenced
(b) Never get data not used
(c) Never prematurely replace other data
(d) Never interfere with other cache activity

Item (a) conflicts with (b), (c) and (d)
Prefetching Policy

Policy
  What to prefetch?
  When to prefetch?

Simplest Policy

Enhancements
Prefetching Policy

Policy

What to prefetch?
When to prefetch?

Simplest Policy

One block (spatially) ahead on every access

Enhancements
Prefetching Policy

Policy

What to prefetch?
When to prefetch?

Simplest Policy

One block (spatially) ahead on every access

Enhancements

On every miss
Because hard to determine on every reference whether block to be prefetched is already in cache
Detect stride and prefetch on stride
Prefetch into prefetch buffer
Prefetch more than 1 block (for instruction caches, when block small)
Software Prefetching

Use compiler to

Prefetch early

E.g., one loop iteration ahead

Prefetch accurately
Software Prefetching Example

for (i = 0; i < N-1; i++) {
    \ldots = A[i]
    /* computation */
}

Assume each iteration takes 10 cycles with a hit, memory latency is 100 cycles
Software Prefetching Example

```c
for (i = 0; i < N-1; i++) {
    ... = A[i]
    /* computation */
}
```

Assume each iteration takes 10 cycles with a hit, memory latency is 100 cycles

```c
for (i = 0; i < N-1; i++) {
    prefetch(A[i+10])
    ... = A[i]
    /* computation */
}
```
for (i = 0; i < N-1; i++) {
    ... = A[i]
    /* computation */
}

Assume each iteration takes 10 cycles with a hit, memory latency is 100 cycles
for (i = 0; i < N-1; i++) {
    prefetch(A[i+10])
    ... = A[i]
    /* computation */
}

This is good with one word cache blocks
for (i = 0; i < N-1; i++) {
    ... = A[i]
    /* computation */
}

Assume each iteration takes 10 cycles with a hit, memory latency is 100 cycles, cache block is two words

Changes?
for (i = 0; i < N-1; i++) {
    prefetch(A[i+10])
    ... = A[i]
    /* computation */
}
Software Prefetching Example**

for (i = 0; i < N-1; i++) {
    … = A[i]
    /* computation */
}

Assume each iteration takes 10 cycles with a hit, memory latency is 100 cycles, cache block is two words

for (i = 0; i < N-1; i+=2) {
    prefetch(A[i+10])
    … = A[i]
    … = A[i+1]
    /* computation for two iterations */
}
Restructure so that operations on a cache block done before going to next block

do i = 1 to rows
    do j = 1 to cols
        sum = sum + x[i,j]

What is the cache behavior?
Software Restructuring (Cont.)

do i = 1 to rows
  do j = 1 to cols
    sum = sum + x[i, j]

Column major order in memory

Code access pattern

Better code??

Called loop interchange
Many such optimizations possible (merging, fusion, blocking)
Software Restructuring (Cont.)

\[
\text{do } i = 1 \text{ to rows} \\
\quad \text{do } j = 1 \text{ to cols} \\
\qquad \text{sum } = \text{sum } + \ x[i,j]
\]

Column major order in memory

\[x[i,j], x[i+1,j], x[i+2,j], \ldots\]

Code access pattern

Better code??

Called loop interchange

Many such optimizations possible (merging, fusion, blocking)
do i = 1 to rows
  do j = 1 to cols
    sum = sum + x[i,j]

Column major order in memory
  x[i,j], x[i+1,j], x[i+2,j], ...

Code access pattern
  x[1,1], x[1,2], x[1,3], ...

Better code??

Called loop interchange
Many such optimizations possible (merging, fusion, blocking)
do i = 1 to rows
  do j = 1 to cols
    sum = sum + x[i,j]

Column major order in memory
  x[i,j], x[i+1,j], x[i+2,j], ...

Code access pattern
  x[1,1], x[1,2], x[1,3], ...

Better code

do j = 1 to cols
  do i = 1 to rows
    sum = sum + x[i,j]

Called loop interchange
Many such optimizations possible (merging, fusion, blocking)
Pipelining and Banking for Higher Bandwidth

Pipelining

Old: cache access = 1 cycle

New: 1 cycle caches would slow the whole processor

   Pipeline: cache hit may take 4 cycles (affects misspeculation penalty)

Multiple banks

   Block based interleaving allows multiple accesses per cycle
Handling Writes - Pipelining

Writing into a writeback cache

  Read tags (1 cycle)
  Write data (1 cycle)

Key observation

  Data RAMs unused during tag read
  Could complete a previous write

Add a special ``Cache Write Buffer'' (CWB)

  During tag check, write data and address to CWB
  If miss, handle in normal fashion
  If hit, written data stays in CWB

When data RAMs are free (e.g., next write) store contents of CWB in data RAMs.

  Cache reads must check CWB (bypass)

Used in VAX 8800
Handling Writes - Write Buffers

Writethrough caches are simple
  But 5-15% of all instructions are stores
  Need to buffer writes to memory

Write buffer
  Write result in buffer
  Buffer writes results to memory
  Stall only when buffer is full
  Can combine writes to same line (Coalescing write buffer - Alpha)
  Allow reads to pass writes

What about data dependencies?
  Could stall (slow)
  Check address and bypass result
Writeback caches need buffers too

10-20% of all blocks are written back

10-20% increase in miss penalty without buffer

On a miss

Initiate fetch for requested block
Copy dirty block into writeback buffer
Copy requested block into cache, resume CPU
Now write dirty block back to memory