### Chapter 2: Memory Hierarchy Design (Part 2)

- Introduction
- Caches
  - Review of basics
  - Advanced methods
- Main Memory
- Virtual Memory

### Advanced Cache Design

- 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

<table>
<thead>
<tr>
<th>Method 1: Hardware Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advantages</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td>Disadvantages</td>
</tr>
</tbody>
</table>
Method 2: Analytic Models

Mathematical expressions

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

Disadvantages

Trace-Driven Simulation

Step 1:
Program + Input Data → Execute and Trace → 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?
Average Memory Access Time and Performance

What About Non-Performance Metrics?
Area, power, detailed timing
CACTI for caches
McPAT: microarchitecture model for full multicore

Timing Data from CACTI

Power Data from CACTI

Copyright © 2011, Elsevier Inc. All rights Reserved.
**Multilevel Caches**

![Multilevel Caches Diagram](image)

**Why Multilevel Caches?**

**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, cont.**

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 \( \frac{L2\text{ blocksize}}{L1\text{ blocksize}} \)
**Multilevel Exclusion**

What if the L2 cache is only slightly larger than L1?
Multilevel exclusion => A line in L1 is never in L2 (AMD Athlon)

**Level Two Cache Design**

L1 cache design similar to singlelevel 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 MBytes)
BUT: L2 affects miss penalty, L1 affects clock rate

**Level Two Cache Example**

Recall adding associativity to a singlelevel cache helped performance if
\[ \Delta_{cache} + \Delta_{miss} \times t_{memory} < 0 \]
\[ \Delta_{miss} = \frac{1}{2}\% \times 20 \text{ cycles} \implies \Delta_{cache} \ll 0.1 \text{ cycle} \]
Consider doing the same in an L2 cache, where
\[ t_{avg} = t_{cache1} + miss1 \times t_{cache2} + \text{global-miss2} \times t_{memory} \]
Improvement only if
\[ \text{miss1} \times \Delta_{cache2} + \text{miss2} \times t_{memory} < 0 \]
\[ \Delta_{cache2} < \frac{\text{miss2} \times t_{memory}}{0.005} \]
\[ \Delta_{cache2} < 100 = 1 \text{ cycle} \]

**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

**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 8 cycles before delivering two words per cycle, we assumed
\[ t_{memory} = t_{access} + B \times t_{transfer} = 8 + B \times 1/2 \]
where B is block size in words
How can we do better?
Reducing Miss Cost, cont.

\[ t_{\text{memory}} = t_{\text{access}} + B \times t_{\text{transfer}} = 8 + B \times 1/2 \]

\( \Rightarrow \) the whole block is loaded before data returned

If main memory returned the reference first (requestedwordfirst) and the cache returned it to the processor before loading it into the cache data array (fetchbypass, early restart),

\[ t_{\text{memory}} = t_{\text{access}} + MB \times t_{\text{transfer}} = 8 + 2 \times 1/2 \]

where \( MB \) 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

LockupFree Caches

Normal cache stalls while a miss is pending

LockupFree 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

LockupFree 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

What state do we need in MSHR?
**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

**Beyond Simple Blocks, cont.**

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 Policy**

Policy
- What to prefetch?
- When to prefetch?

Simplest Policy
- ?

Enhancements
**Software Prefetching**

Use compiler to
- Prefetch early
  - E.g., one loop iteration ahead
- Prefetch accurately

**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

**Software Prefetching Example**

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

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

**Software Restructuring**

Restructure so that operations on a cache block done before going
to next block

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

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

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

Column major order in memory

Code access pattern

Better code??

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 missspeculation 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
Handling Writes - Writeback Buffers

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

Usually only need 1 or 2 writeback buffers