Cache Optimization

Today's topics:

Look at memory access times

- improve hit time
- reduce miss rate
- reduce miss penalty

Full disclosure

- I'm winging this one
- more detail on white board than slides

Optimize What

- Basic quantitative metric

$$\text{AverageMemoryAccessTime} = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

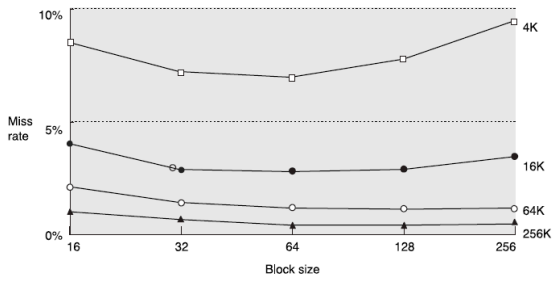
  - but in a parallel world it's about exposed latency

$$\frac{\text{MemoryStallCycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{TotalMissLatency} - \text{OverlappedMissLatency})$$

  - note that miss penalty
    - » f(transfer_rate/bandwidth, latency_next_lower_cache)
    - » so improve bandwidth helps
- Optimize
  - reduce hit time (Amdahl's law – it's the common case)
  - increase cache bandwidth
  - reduce miss penalty
  - reduce miss rate
  - increase overlap

Knee of Curve Problem

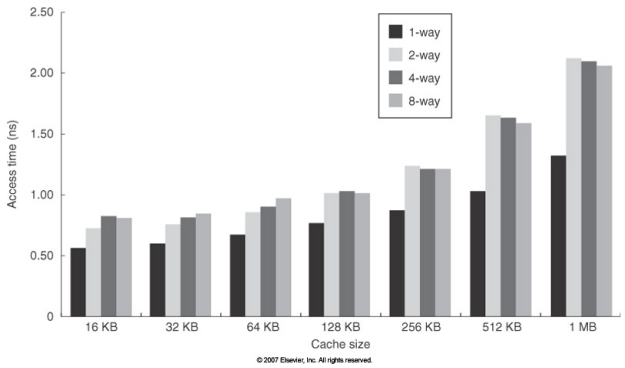
- Bigger problem for small caches – e.g. L1
  - latency vs. transfer time



Reduce Hit Time

- Small and simple caches
  - keep cache hierarchy on chip
    - » off chip access is 10-100x penalty
  - small, direct-mapped
    - » note L1$ size doesn't change much w/ technology
    - » L2 is where the biggest change occurs
  - associativity is a double edged sword
- Next slide
  - models based on CACTI
    - » common research tool
    - » book model is CACTI-IV
      - note this doesn't accurately deal w/ wire delay
      - current version is 6.5
        - much better wire models – SPICE back annotated validation
    - » undecided
      - might be used in HW4

Hit Time Effects



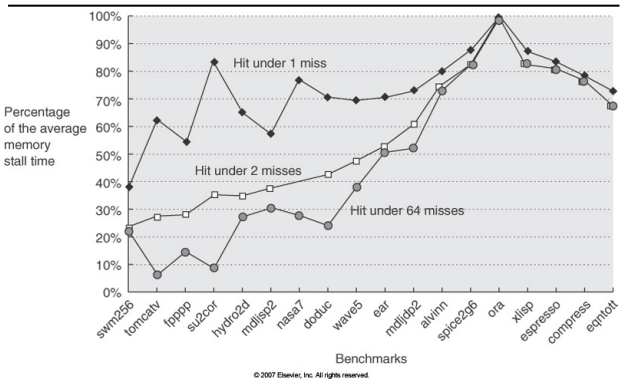
Hit-time Improvement #2

- Way prediction
  - basics covered last time
    - » saves comparator power
    - » increased associativity of set-assoc caches
      - reduces conflict misses
      - but way prediction give performance of direct mapped cache
  - tactics similar to branch prediction
    - » PC of load or store
      - keep track of which way was hit
        - very similar to local Gselect predictor
      - keep track of progression
        - similar to stride prediction
- Practice
  - way prediction used in both MIPS and Pentium IV processors
- Prediction accuracy
  - ~85%

Increase Cache Bandwidth

- Pipelined caches
  - Just like processor
    - » pipelining can increase latency
    - » fill and spill penalty when things go wrong
  - throughput improves on average
    - » note L1 latency increasing
      - 3'ish cycles today but launch a new access every cycle
- Non-blocking caches
  - for when L1+ misses occur
  - MSHR's and dynamic issue logic

Non-blocking Cache Data



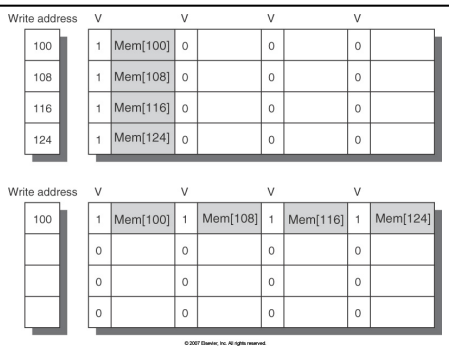
Multi-Banked Caches

- Interleave for improved bandwidth
  - practice
    - » Opteron – 2 banks
    - » Sun Niagra – 4 banks
  - Idea
    - » no conflict accesses issues faster than bank latency
    - » alternative to hit-under-miss & MSHR solution
    - » banks are smaller so latency is reduced
- Bigger advantage w/ multi-core and shared L2
  - downside
    - » faster bus
    - » OR multiple buses
    - » note:
      - long wires on buses inherently induce problems
        - slower performance – C effect
        - high power – signal integrity
          - pre- and post-emphasis (e.g. QPI, HT, dynamic balance)

Reduce Miss Penalty

- Critical word first
  - overlap transfer with ability to use data
  - complicates next level access
    - » not all that bad
    - » at DRAM level
      - MEM_CTL in the way anyway
      - overhead to support is minor
- Coalesce/Combining/Merge Write Buffer
  - writes happen from a register value
  - cache lines bigger
    - » so buffer writes by cache line
    - » same unit of transfer
      - cache to cache
      - cache to DRAM

Combining Write Buffer Example



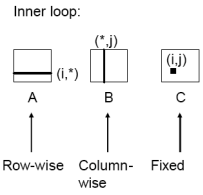
What are the cost effects?

Reduce Miss Rate

- Compiler optimizations
  - compiler knows cache organization
    - » arrange code and data to minimize misses
  - loop interchange – improves spatial locality
    - » walk cache line in inner loop
      - e.g. matrix multiply is the canonical example
  - blocking – improves spatial locality
    - » put code into phases
      - do as much as you can on this data frame before moving to next frame
      - avoids register spill and refill as well as cache misses
    - » matrix multiply again

Matmul Example ijk

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```



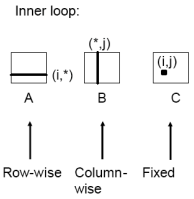
- Miss RATIO per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

1/m where m is number of elements in a cache line

Loop Interchange to jik

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

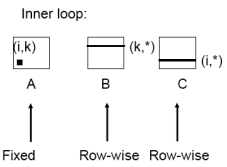


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Loop Interchange to kij

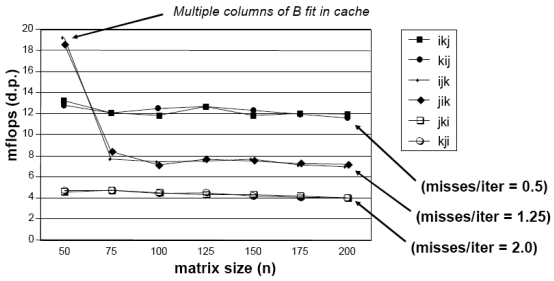
```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

All Possibilities



Blocked MatMul Example

Example: N = 8; *sub-block size* = 4

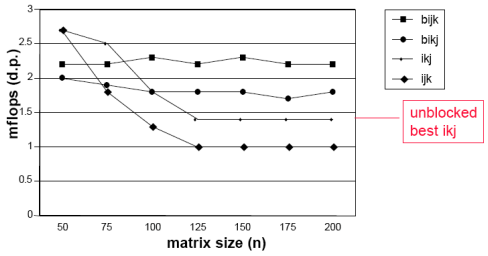
$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \times \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

Key idea: Sub-blocks (i.e., $\mathbf{A}_{xy}$) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Blocked MatMul Performance



Others

- Prefetch
  - reduces miss penalty and miss rate
    - » If done right
    - » added complexity, power, and screw up potential
      - discussed last lecture
  - can be done either by HW or SW
- Next level cache
  - reduces miss penalty
    - » In best case
  - increases miss penalty
    - » In worst case
    - » "swing to miss" principle

Ancillary Caches

- Victim cache (Jouppi)
  - small cache to hold victimized lines
  - idea allows arbitrary associativity for small number of lines
    - » total extra associativity = size of victim cache
  - downside
    - » parallel check of regular and victim
    - » fully associative
- Trace cache (Weiser, Peleg)
  - Intel P4
    - » expensive – many instruction copies
- Assist cache (HP and somebody you know)
  - 1st touch goes to assist
  - 2nd touch goes to regular cache
    - » makes prefetch less likely to contaminate cache
  - downside
    - » similar to victim cache

Summary I

Technique	Miss Rate	Miss Penalty	Hit Time	HW Complexity	Comments
Larger Cache	win		lose	easy	cost is approx. linear
Larger Block Size	win	lose		easy	trivial engineering effort
Higher Associativity	win		lose	1	associative match isn't free
Victim Caches	win			2	e.g. HP 7200
Pseudo-Associative	win			2	Used in L2 of MIPS R10000
HW Prefetch of I and D	win			2	D fetch hard to do correctly
Compiler controlled prefetch	win			3	Needs non-blocking cache
Compiler cache scheduling (blocking, ...)	win			0	Too bad it's hard to do for all applications - loop focus for now

Summary II

Technique	Miss Rate	Miss Penalty	Hit Time	HW Complexity	Comments
Prioritizing Read Misses		win		1	write buffer - simple
Subblock placement		win		1	good at reducing tag overhead
Early restart + critical word first		win		2	MIPS R10K, IBM 620
Nonblocking Caches		win		3	MIPS R10K, AXP 21064
Second Level Caches		big win		2	big additional cost for all that SRAM
Small Simple Caches	lose		win	0	trivial so it's widely used
Avoid translation effects			win	2	AXP 21064 several PA-whatevers
Pipelining Writes			win	1	AXP 21064, PA-8000, UltraSPARC

Conclusion

- **Cost**
  - **focus on HW cost**
    - » **compiler viewed as free if you're a HW geek**
  - **low**
    - » **small caches, way-prediction, pipelined cache access, banked caches, compiler tactics**
  - **medium**
    - » **critical word first and early restart**
    - » **instruction prefetch**
      - **access is more regular**
    - » **victim and assist caches**
  - **expensive**
    - » **trace caches**
      - **now that power is a big issue**
    - » **data prefetch**
      - **access irregular → wasted speculation**