

CS422

Computer Architecture

Spring 2004

Lecture 19, 04 Mar 2004

Bhaskaran Raman
Department of CSE
IIT Kanpur

<http://web.cse.iitk.ac.in/~cs422/index.html>

Topics for Today

- Cache Performance
- Cache Misses: The Three C's
- Improving the Cache Miss Rate
-
- **Scribe?**

Cache Performance

- Miss rate is an important metric
 - But not the only one

Avg. mem. access time =

Hit time + Miss rate $\times$ Miss penalty

- Hit time, Miss penalty can be expressed
 - In absolute terms,
 - Or, in terms of number of clock cycles
- Miss rate decrease may imply reduced performance
 - Example: unified vs. split cache

CPU Performance, with Cache

$$CPU\ time = (CPU\ cycles + Mem.\ stall\ cycles) \times Cycle\ time$$

$$Mem.\ stalls = Reads \times Read\ miss\ rate \times Read\ miss\ penalty \\ + Writes \times Write\ miss\ rate \times Write\ miss\ penalty$$

$$CPU\ time = IC \times Cycle\ time \times \\ (CPI + \frac{Mem.\ accesses}{instn.} \times Miss\ rate \times Miss\ penalty)$$

Effect of Cache on Performance

- Some typical values:
 - $\text{CPI} = 1$
 - Mem. per instn. = 1.35
 - Miss rate = 2%
 - Miss penalty = 50
- Mem. stalls comparable to CPI!
 - Cache behaviour is an important component of performance
 - More important for lower CPI

Improving Cache Performance

Avg. mem. access time =

Hit time + Miss rate $\times$ Miss penalty

- Three possibilities:
 - Reduce miss rate
 - Reduce miss penalty
 - Reduce hit time
- Beware of slowing down the CPU!
- Example:
 - Set associative ==> potentially higher cycle time

Cache Misses: The Three C's

- **Compulsory:** first access to a block
 - Also called cold start, or first reference misses
- **Capacity:** misses due to cache being small
- **Conflict:** two memory blocks mapping onto the same cache block
 - Also called collision, or interference misses

Cache size	Associativity	Compulsory	Capacity	Conflict	Total	Frac. Compulsory	Frac. Capacity	Frac. Conflict
1KB	1-way	0.20%	8.00%	5.20%	13.40%	0.01	0.6	0.39
1KB	2-way	0.20%	8.00%	2.30%	10.50%	0.02	0.76	0.22
1KB	4-way	0.20%	8.00%	1.30%	9.50%	0.02	0.84	0.14
1KB	8-way	0.20%	8.00%	0.50%	8.70%	0.02	0.92	0.06
2KB	1-way	0.20%	4.40%	5.20%	9.80%	0.02	0.45	0.53
2KB	2-way	0.20%	4.40%	3.00%	7.60%	0.03	0.58	0.39
2KB	4-way	0.20%	4.40%	1.80%	6.40%	0.03	0.69	0.28
2KB	8-way	0.20%	4.40%	0.80%	5.40%	0.04	0.81	0.15
4KB	1-way	0.20%	3.10%	3.90%	7.20%	0.03	0.43	0.54
4KB	2-way	0.20%	3.10%	2.40%	5.70%	0.04	0.54	0.42
4KB	4-way	0.20%	3.10%	1.60%	4.90%	0.04	0.63	0.33
4KB	8-way	0.20%	3.10%	0.60%	3.90%	0.05	0.79	0.15
8KB	1-way	0.20%	2.30%	2.10%	4.60%	0.04	0.5	0.46
8KB	2-way	0.20%	2.30%	1.30%	3.80%	0.05	0.61	0.34
8KB	4-way	0.20%	2.30%	1.00%	3.50%	0.06	0.66	0.29
8KB	8-way	0.20%	2.30%	0.40%	2.90%	0.07	0.79	0.14
16KB	1-way	0.20%	1.50%	1.20%	2.90%	0.07	0.52	0.41
16KB	2-way	0.20%	1.50%	0.50%	2.20%	0.09	0.68	0.23
16KB	4-way	0.20%	1.50%	0.30%	2.00%	0.1	0.75	0.15
16KB	8-way	0.20%	1.50%	0.20%	1.90%	0.11	0.79	0.11
32KB	1-way	0.20%	1.00%	0.80%	2.00%	0.1	0.5	0.4
32KB	2-way	0.20%	1.00%	0.20%	1.40%	0.14	0.71	0.14
32KB	4-way	0.20%	1.00%	0.10%	1.30%	0.15	0.77	0.08
32KB	8-way	0.20%	1.00%	0.10%	1.30%	0.15	0.77	0.08
64KB	1-way	0.20%	0.70%	0.50%	1.40%	0.14	0.5	0.36
64KB	2-way	0.20%	0.70%	0.10%	1.00%	0.2	0.7	0.1
64KB	4-way	0.20%	0.70%	0.00%	0.90%	0.22	0.78	0
64KB	8-way	0.20%	0.70%	0.00%	0.90%	0.22	0.78	0
128KB	1-way	0.20%	0.40%	0.40%	1.00%	0.2	0.4	0.4
128KB	2-way	0.20%	0.40%	0.10%	0.70%	0.29	0.57	0.14
128KB	4-way	0.20%	0.40%	0.10%	0.70%	0.29	0.57	0.14
128KB	8-way	0.20%	0.40%	0.00%	0.60%	0.33	0.67	0

Reducing Cache Misses

- Capacity: increase cache size
 - Thrashing can happen otherwise
- Conflict: increase associativity
 - But, greater complexity, slower hit time
- Compulsory: increase block size
 - But, greater miss penalty!

Technique-1: Larger Blocks

- Reduces compulsory misses
 - By improving spatial locality
- Increases miss penalty
- Also, may increase conflict/capacity misses

Cache size	1KB	4KB	16KB	64KB	256KB
Block size					
16B	15.05%	8.57%	3.94%	2.04%	1.09%
32B	13.34%	7.24%	2.87%	1.35%	0.70%
64B	13.76%	7.00%	2.64%	1.06%	0.51%
128B	16.64%	7.78%	2.77%	1.02%	0.49%
256B	22.01%	9.51%	3.29%	1.15%	0.49%

Larger Blocks (continued)

- Miss penalty depends on:
 - Memory latency, memory bandwidth
- Assuming latency of 40 cycles, and bandwidth of 16 bytes per 2 cycles, AMAT values are:

Cache size		1KB	4KB	16KB	64KB	256KB
Block size	Miss penalty					
16B	42	7.32	4.6	2.66	1.86	1.46
32B	44	6.87	4.19	2.26	1.59	1.31
64B	48	7.61	4.36	2.27	1.51	1.25
128B	56	10.31	5.35	2.55	1.57	1.27
256B	72	16.85	7.84	3.37	1.83	1.35

Technique-2: Higher Associativity

- Reduces conflict misses
- But, increases hit time
- 8-way as good as fully associative
- Rule of thumb:
 - Direct mapped cache of size N has the same miss rate as a 2-way cache of size $N/2$

Technique-3: Victim Cache

- Small cache of “victim” blocks, which were thrown out recently
 - Fully associative
- Reduces conflict misses
- Does not affect cycle time, or miss penalty
- Study: 4-entry victim cache removed 20-95% of conflict misses in a 4KB direct mapped cache

Technique-4: Pseudo-Associative Cache

- Also called column associative
- Hit proceeds just as in a direct-mapped cache
- Miss ==> check in set (by flipping MSB of index)
- May need to swap contents in the set

$$Miss\ rate_{pseudo} = Miss\ rate_{2-way}$$

$$Miss\ penalty_{pseudo} = Miss\ penalty_{1-way}$$

$$Hit\ time_{pseudo} = Hit\ time_{1-way} + Alt.\ hit\ rate \times k$$

$$Alt.\ hit\ rate = Miss\ rate_{1-way} - Miss\ rate_{2-way}$$

Technique-5: Hardware Prefetching

- Fetch more than required, on a miss
 - Prefetch into cache, or another small buffer (faster than memory)

Avg. mem. access time =

Hit time + Miss rate $\times$ Prefetch hit rate $\times k$

+ Miss rate $\times$ Prefetch miss rate $\times$ Miss penalty

Technique-6: Compiler Controlled Prefetch

- Special instructions for prefetching data
 - Non-faulting instructions are most useful
 - CPU should be able to proceed in parallel with cache
 - Non-blocking cache
- Example:

```
for (i = 0; i < 3; i++) {  
    for(j = 0; j < 100; j++) {  
        a[i][j] = b[j][0] * b[j+1][0];  
    }  
}
```

Technique-7: Compiler Optimizations

- Merging arrays

```
int val[1000];      ────>  struct merge { int val; int key; };  
int key[1000];      struct merge M[1000];
```

- Improves spatial locality

- Loop interchange

```
for(j = 0; j < 100; j++)  
    for(i = 0; i < 100; i++) ────>    for(i = 0; i < 100; i++)  
        x[i][j] = 0;                    for(j = 0; j < 100; j++)  
                                         x[i][j] = 0;
```

- Improves spatial locality

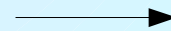
Compiler Optimizations (continued)

- Loop fusion

```
for(i = 0; i < 100; i++)
```

```
    for(j = 0; j < 100; j++)
```

```
        a[i][j] = b[i][j] + c[i][j];
```



```
for(i = 0; i < 100; i++)
```

```
    for(j = 0; j < 100; j++)
```

```
        a[i][j] = b[i][j] + c[i][j];
```

```
        d[i][j] = 2*a[i][j];
```

```
for(i = 0; i < 100; i++)
```

```
    for(j = 0; j < 100; j++)
```

```
        d[i][j] = 2*a[i][j];
```

- Improves temporal locality

- Blocking: operate on small blocks of matrices

- Improves temporal locality

Miss-Rate Reduction: Summary

- Larger blocks
- Higher associativity
- Victim cache
- Pseudo-associativity
- Hardware prefetching
- Software controlled prefetching
- Code optimization by compiler