COL216 - COMPUTER ARCHITECTURE

Prof. Kolin Paul

ANUP LAL NAYAK, 2022CS51827

PROJECT : CACHE SIMULATOR (github)

INTRODUCTION:

Caches play a crucial role in enhancing the performance of computer systems by reducing the time taken to access frequently used data. However, designing an efficient cache configuration involves balancing various factors such as hit rates, miss penalties, and total cache size. In this report, I analyze different cache configurations using memory traces and a cache simulator to determine the most effective configuration.

EXPERIMENT DESIGN:

To evaluate cache configurations, I configured the cache simulator with different parameters, including the number of sets, blocks per set, block size, write-allocate, write-back/write-through and eviction policy (FIFO or LRU) and obtained the cycles, hit-rate and miss-penalties for all possible configurations for these trace files:
1. swim.trace
2. gcc.trace

I ran these configurations for three cache-types (sets,blocks,bytes):
1. (256;64;16) -> mid-associative
2. (64;128;32) -> high-associative
3. (1;1024;256) -> fully-associative

Why?- The size of cache (total number of bytes) is same in all 3 and hence the cache overhead for each of them is not very different.
This is the log file containing all relevant data collected : 216logfile .
After obtaining all the data, I observed the plots for various cache-parameters against miss-penalties, cycles and hit-rate.

CONCLUSION & BEST CACHE :

While optimal cache-parameters depend on the program structure and use-case. In general, the best cache configuration comprises the following parameters according to my understanding and simulation of my cache:

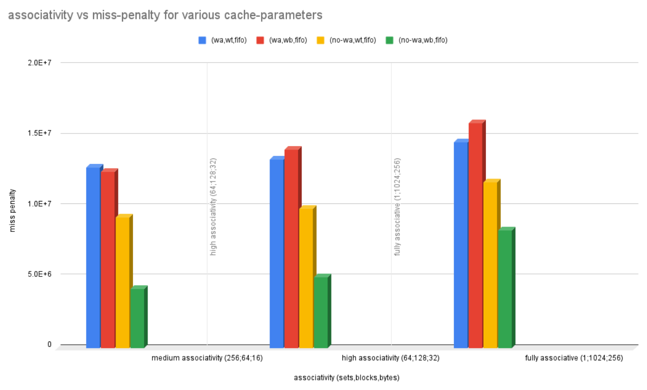
- **Write Allocation Strategy**: Write-allocate
- **Write Policy**: Write-back
- **Eviction Policy**: FIFO
- **Associativity**: High Associative but not fully-associative.

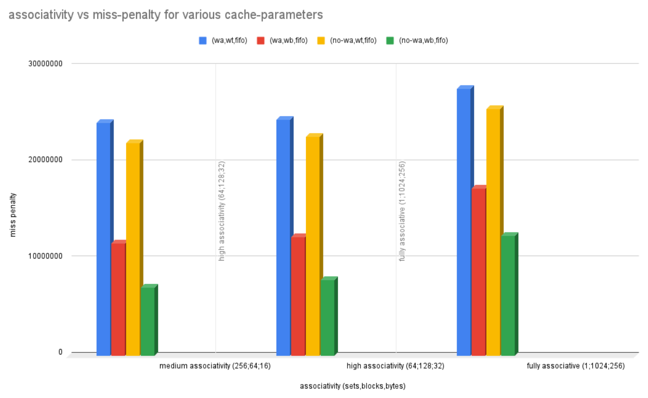
This configuration optimizes cache performance by minimizing memory traffic, efficiently handling write operations, and achieving higher hit rates while maintaining reasonable hardware complexity and cost-effectiveness.

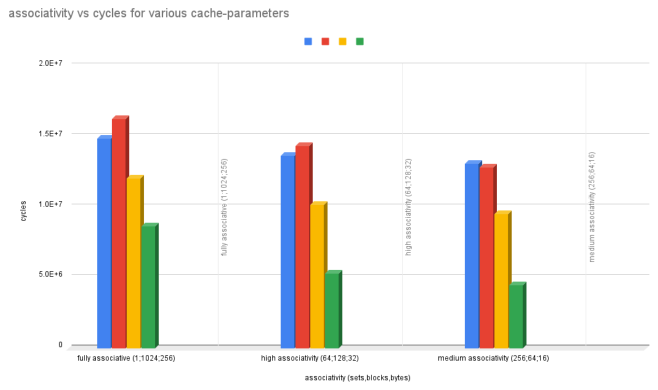
OBSERVATION:

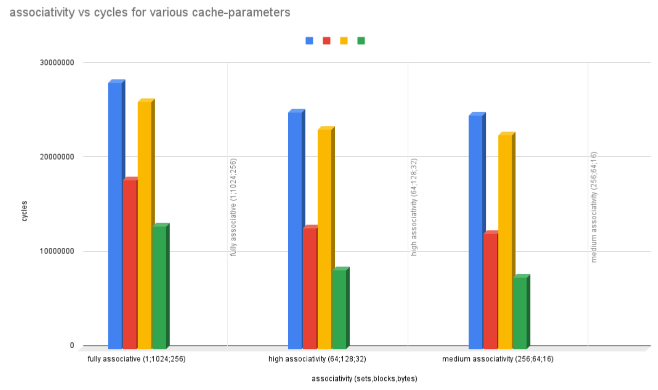
1. swim.trace

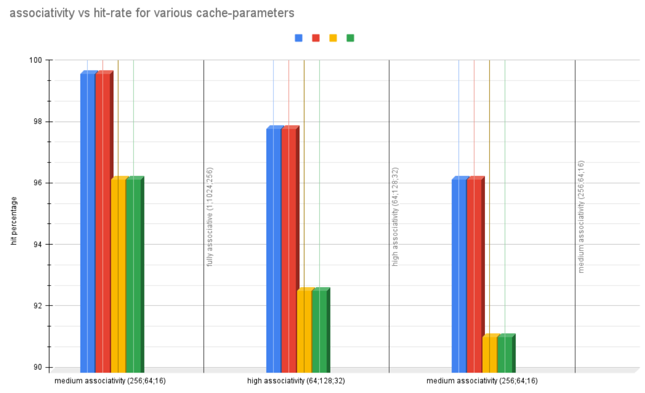
2. gcc.trace

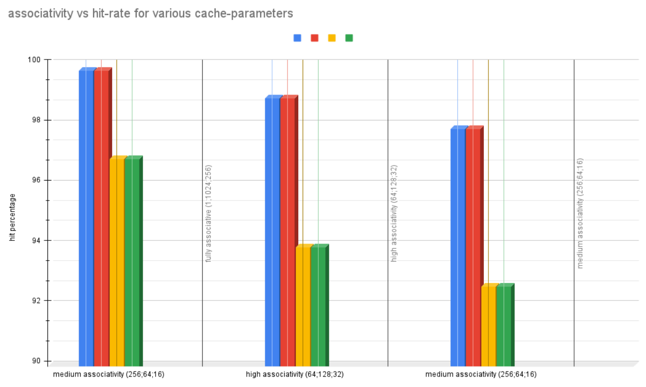












EXPERIMENT RESULTS:

The experiments revealed significant variations in cache performance based on different configurations.

- No-write-allocate takes a lower number of cycles but has significantly lesser hit-rate as compared to write-allocate.

avg across all data points	**write-allocate**	**no-write-allocate**
cycles	1,57,97,871.33	1,26,14,113
hit-rate	98.368%	93.872%

- In no-write-allocate, configuration write-back took significantly fewer cycles compared to write-through. But in write-allocate configuration, the cycles were similar with write-back taking a little more than write-through.
- Furthermore, I found that the choice of eviction policy did not have a notable impact on cache effectiveness. But, hardware implementation of LRU is extremely costly and unfeasible hence FIFO is clearly the better eviction policy.
- By increasing associativity, the number of cycles increases but the hit-rate also increases.

avg across all data points	**mid-associative**	**high-associative**	**fully-associative**
cycles	1,34,04,725.5	1,41,44,550.5	1,71,58,463
hit-rate	94.433%	95.803%	98.122%

ARGUMENTS SUPPORTING MY RESULTS:

1. Write Allocation Strategy:
    - Lower Number of Cycles: Write-allocate likely results in lower cycles due to the reduction in memory traffic. By bringing entire blocks into cache for write operations, subsequent read or write operations to the same block incur fewer cycles, as the data is already in cache.
    - Slight Hit-Rate Reduction: The slightly lower hit rate in the write-allocate strategy could be attributed to the overhead of bringing in entire blocks for write operations, which might displace useful data that would otherwise remain in cache with a no-write-allocate approach.
2. Write Policy:
    - Write-Back Efficiency: Write-back policy minimizes memory writes by only updating the cache and delaying writes to main memory until the cache block is evicted. This reduces memory traffic and, consequently, the number of cycles required for memory operations, particularly noticeable in no-write-allocate configurations.
    - Similar Performance in Write-Allocate: In the write-allocate scenario, the performance difference between write-back and write-through policies is minimal. This could be because the write-allocate strategy already optimizes write operations by bringing in entire blocks into cache.
3. Eviction Policy:
    - Cost-Effectiveness of FIFO: FIFO eviction policy, while simplistic, is efficient and easy to implement in hardware. LRU, while theoretically optimal, often requires complex hardware implementations such as tagged comparators for tracking access history, resulting in increased cost and complexity.
4. Associativity:
    - Higher Hit Rates with Increased Associativity: Increasing associativity allows for more flexibility in storing data and reduces the likelihood of cache conflicts, leading to higher hit rates. However, this comes at the cost of increased hardware complexity and longer access times due to the additional comparisons required.

BIBLIOGRAPHY:

H. Ma, X. Yue and J. Song, "Cache Performance Simulation and Analysis under Multi-parameters," 2010 International Conference on Machine Vision and Human-machine Interface, Kaifeng, China, 2010