UiT The Arctic University of Norway

Department of Computer Science

Cache Simulator

Max Roness Hovding, UiT ID mho251@uit.no, GitHub username: Maxhovding

GitHub repo: cache-simulator-m

INF-2200

1 Introduction

This benchmark evaluates cache configurations, for a bubble sort algorithm. This is an algorithm with a clear memory access pattern. Bubble sort was selected due to its simplicity and because it is an algorithm that repeatedly accesses and modifies in-place data.

The data set we are testing is an array of 10000 randomly generated integers, to mimic a realistic scenario. This range should ensure a comprehensive enough size to test the cache behaviour.

Memory Access Pattern

The memory access pattern of a bubble sort is largely sequential, as it compares and swaps elements in the array. The algorithm mainly performs read operations, as it scans through the array, and performs write operations, when swaps are necessary. The pattern and frequency of course depend on the array that is randomly generated, and the result could be different from array to array. There is also a frequent re-access of elements during sorting, and an accessing of elements in close succession.

2 Methodology

Cache Simulator Implementation and Design

The cache simulator is designed to mimic the behavior of a hardware cache. It contains easily configurable parameters for size, block size, associativity and write policy. It includes an L1 Instruction, L1 Data and an L2 unified cache.

Re-implementing the simulator

If you would like to re-implement the simulator to verify the findings of the benchmarks you can follow these steps:

1. **Initialization:**
    - Define cache parameters: size, block size, associativity, and write policy.
    - Allocate memory dynamically for sets and blocks within each cache level.
    - Initialize valid and dirty bits to 0 for all blocks.

2. **Cache Access:**
    - Calculate the index and tag based on the address and cache parameters.
    - Perform a lookup to check if the block with the corresponding tag exists in the cache set.
    - If a hit occurs, update hit counters and potentially mark the block as dirty if it's a write access, according to the write policy.
    - If a miss occurs, select a block for replacement using the LRU policy, update the miss counters, and if necessary, write back the evicted block to the next cache level or memory if the block is dirty.
3. **Memory Operations:**
    - Implement **memory_read** and **memory_write** functions that interface with the cache simulator. These functions should access the L1 cache and, on a miss, proceed to access the L2 cache.
4. **Statistics Gathering:**
    - Maintain hit and miss counters for each cache level and differentiate between read and write operations.
    - Calculate hit ratios for reads, writes, and overall access after the simulation.
5. **Termination:**
    - Deallocate all allocated memory.
    - Output the final hit and miss statistics and hit ratios for each cache.
6. **Counting Cache Hits and Misses**

Cache hits and misses are counted at each cache level:

- **Hits:**
    - Incremented upon finding a block with the matching tag and valid bit set.
    - For write hits under a write-back policy, the dirty bit is set for the corresponding block.
- **Misses:**
    - Incremented when no block with a matching tag is found in the valid set.
    - For misses under write-allocate policy, the new data is fetched into the cache, potentially evicting an existing block (following the LRU policy).

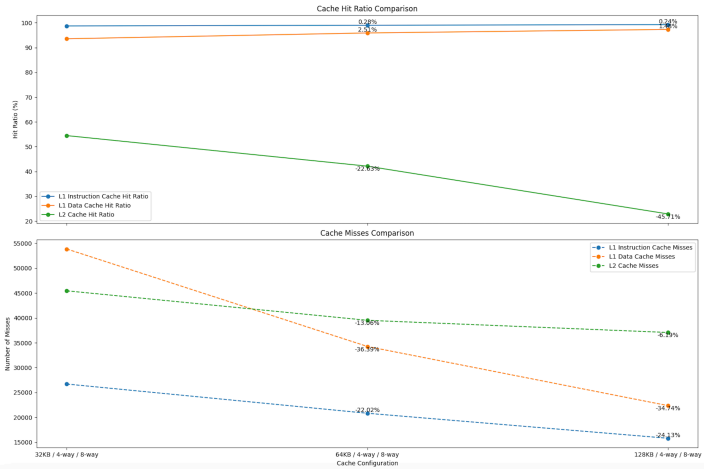
Correctness Test Trace Design

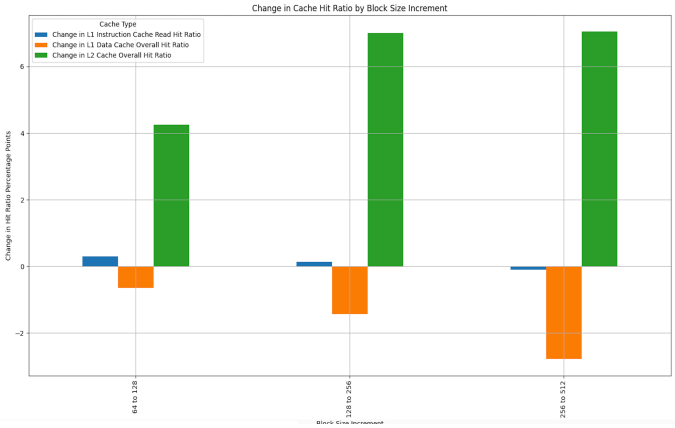
1. **Trace Generation:**
    - Create a trace of memory accesses from a controlled benchmark program.
    - The trace should include a mix of instruction fetches, data reads, and writes, representing realistic access patterns.
2. **Simulator Testing:**
    - Run the trace through the simulator, capturing and logging all cache accesses, hits, and misses.
    - Verify the simulator's decisions on hits, misses, and evictions against expected outcomes based on known access patterns.
3. **Comparing with Expected Results:**
    - Compare the simulator's hit/miss counts and ratios with analytically derived expectations for the given trace.
    - Any discrepancies can be debugged by reviewing the detailed log of cache behaviors.
4. **Parameter Variation:**
    - Run the trace repeatedly for different cache configurations, documenting the impact of each parameter on the performance.
    - Use this data to identify the optimal cache configuration for the given benchmark.

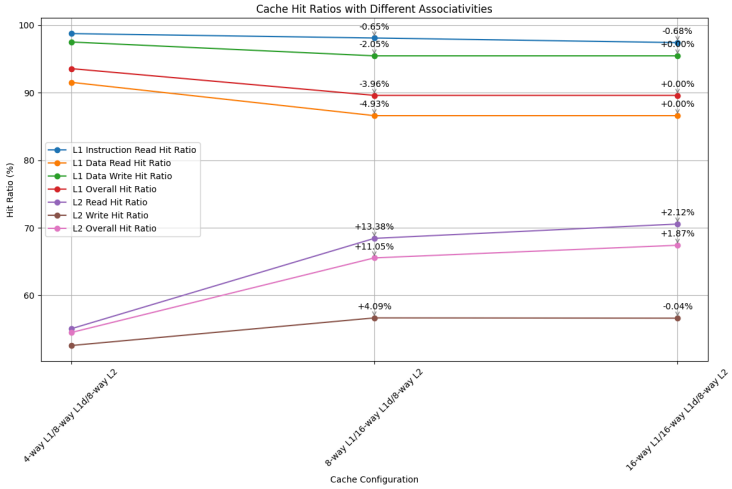
This methodology allows replication of experiments and validation of results, ensuring transparency and correctness in the simulation process.

3 Results

Extensive testing was performed to find the best cache implementation for the benchmark. I have created some graphs highlighting the difference in performance when changing the cache configuration. The graphs are shown down below. The simulator simulates a total of 240,000 instructions, to get a good result.







Graph 1

The first graph, which is in the top left, showcases the difference in performance, when increasing the total cache size. We can see that increasing the L1 cache size, has a clear benefit for their performance. This is also very beneficial for our benchmark, due to the predictable access pattern of bubble sort.

We can see that the hit ratio for L2 decreases when increasing the cache size. This is to be expected. A more effective L1 cache reduces the need for usage of the L2 cache. The L1 cache, with its increased size, can capture more of the workload, and this results in better performance.

It is worth noting that there is a higher hardware cost for increase L1 cache size, due to its speed and location in relation to the CPU. One needs to consider if the increase hardware cost, is worth the performance increase.

Graph 2

The graph in the top right, helps us understand the performance differences, when increasing block size. The data suggest that increasing the block size could be beneficial, but not beyond 128 or 256 bytes. Bubble sort is an algorithm that benefits form larger cache blocks due it being sequential. There is however a point where the block size becomes too large, and further increases do not yield any improved performance.

Graph 3

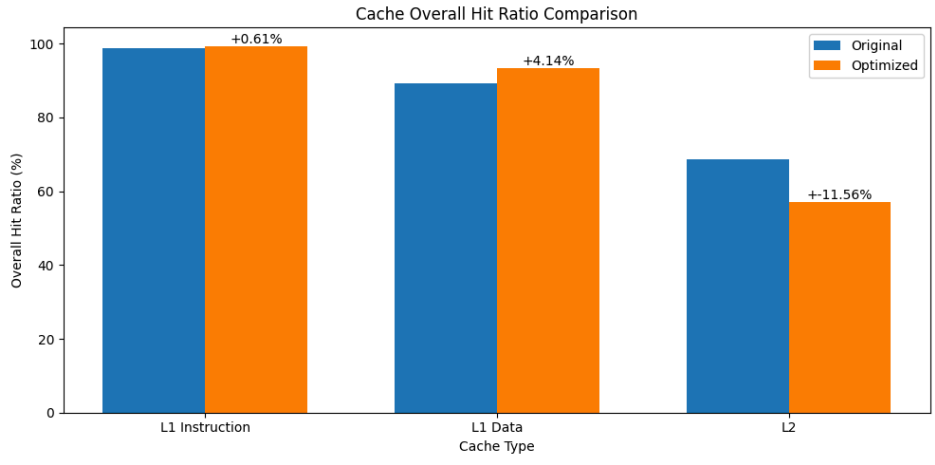
Graph 3 illustrates the changes in performance when changing associativity. We can see that for this benchmark, the increase in associativity does not necessarily yield an increase in performance. We can see that the best performance for the benchmark is recorded when using the original cache associativity specifications. We can therefore conclude that there is not real benefit in increasing associativity for this benchmark.

Change in Write-policy

There were experiments done, to see if changing the write-policies for the caches, would result in an performance increase. The results were not substantial enough to warrant a graph, however I concluded that the best specification would be a write-through for the L1 instruction cache and a write-back for the other two respective caches.

Best result

Through extensive testing, I concluded that the most efficient specification for our cache is a simple increase in cache size for both the L1 instruction cache and the L1 data cache. In the graph below, one can see the increase in performance:



In the above illustration the cache size of L1 instruction was increased to 256 kb and the L1 data cache was increased to 128kb.

4 Discussion

The most interesting result pertains to the relationship between cache associativity and hardware cost. An increase in associativity generally leads to higher hardware complexity and cost due to the need for more comparators and a larger tag index. However, the results indicate that beyond a certain point, increasing associativity for the L1 cache does not proportionately improve the hit ratio for bubble sort, which has a predictable access pattern that benefits more from cache size than from excessive associativity.

The experimentation with different cache sizes indicates that the performance of a cache simulator running a bubble sort benchmark improves significantly with an increase in cache size. Doubling the L1 cache size from 32 KB to 64 KB and then to 128 KB improved the read hit ratio from 91.52% to 94.47% and finally to 96.35% for the L1 Data Cache. This increment also positively impacted the write hit ratio, improving from 97.49% at 32 KB to 99.22% at 128 KB. It's evident that the memory access pattern of bubble sort benefits from larger L1 caches, which reduce the need to access the slower L2 cache.

5 Conclusion

Considering the benchmark results, a larger L1 cache size greatly benefits the sorting algorithm's performance by accommodating its working set more effectively. For the bubble sort benchmark, a larger L1 cache results in significantly fewer L2 accesses, underscoring the importance of sizing the L1 cache to match the application's active data set as closely as possible.

Thus, the optimal cache configuration for a bubble sort benchmark, within the tested parameters, is one with a larger L1 cache, maintaining modest associativity levels. The improved performance of the L1 cache at larger sizes reduces dependency on the L2 cache, which can then be optimized for other tasks or designed with cost-saving measures in mind without significantly affecting the performance of such algorithms. The increase in L1 Data Cache size to 128 KB appears to offer the best balance between performance and potential cost, as it provides a significant improvement in hit ratio and overall efficiency for the simulated bubble sort workload.

6 Sources

ChatGPT was used in the making of this report. ChatGPT was used to analyse test results, and to help write python scripts that could aid in generating graphs.

Hennessy, D. A., & Patterson, J. L. (2021). *Computer Organization And Design MIPS edition.* Elsevier Inc.

University of Washington. (u.d.). *Cache Introduction.* Hentet fra Cache Introduction: https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec15.pdf