Designing an Optimized Cache Strategy for Enhanced Memory Access Efficiency

Ridhima Biswas, Jessica Bohn, Luke Busacca, Nikitha Cherukupalli, Armaan Saleem

Rutgers University – New Brunswick

01:198:211:13 COMPUTER ARCHITECTURE

Dr. Tina Burns

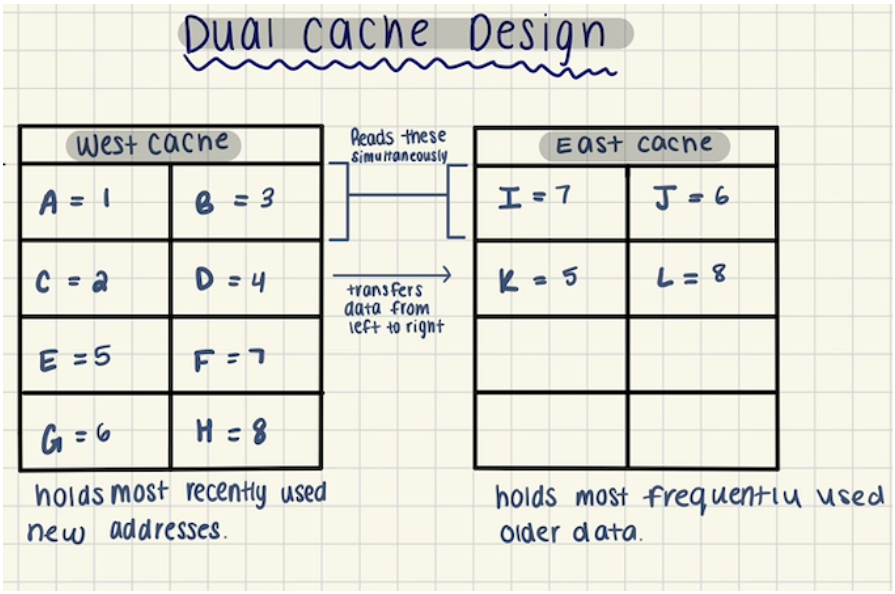
December 14, 2024

Introduction

Cache design plays an important role in modern computer architecture, helping to close the performance gap between powerful CPUs and slower memory systems. Caches are compact, high-speed storage units that sit between the processor and main memory to store frequently accessed data, lowering latency and increasing efficiency. As Bryant and O'Hallaron (2016) state, “the goal of the cache is to reduce the average time to access data from the main memory” (p. 462). Modern cache systems usually have a hierarchical structure, with multiple levels (L1, L2, and L3) varying in size, speed, and location to the CPU. L1 cache is the smallest and fastest, placed closest to the CPU, whereas L3 cache is larger but slower, usually shared by numerous cores.

Cache design is essential for optimizing system performance since it affects processor speed, power consumption, and bandwidth. As Bryant and O'Hallaron (2016) note, “cache performance depends on the cache size, the associativity, and the replacement policy” (p. 463). Cache systems seek to store data that will be reused shortly via temporal and spatial locality. Replacement algorithms, such as Least Recently Used (LRU) and Random Replacement, are used to effectively organize cached data. Effective cache systems decrease the number of memory accesses, resulting in faster data retrieval and greater computing efficiency (Bryant and O'Hallaron, 2016).

Our Strategy: Bilateral Parallel Mapping Cache

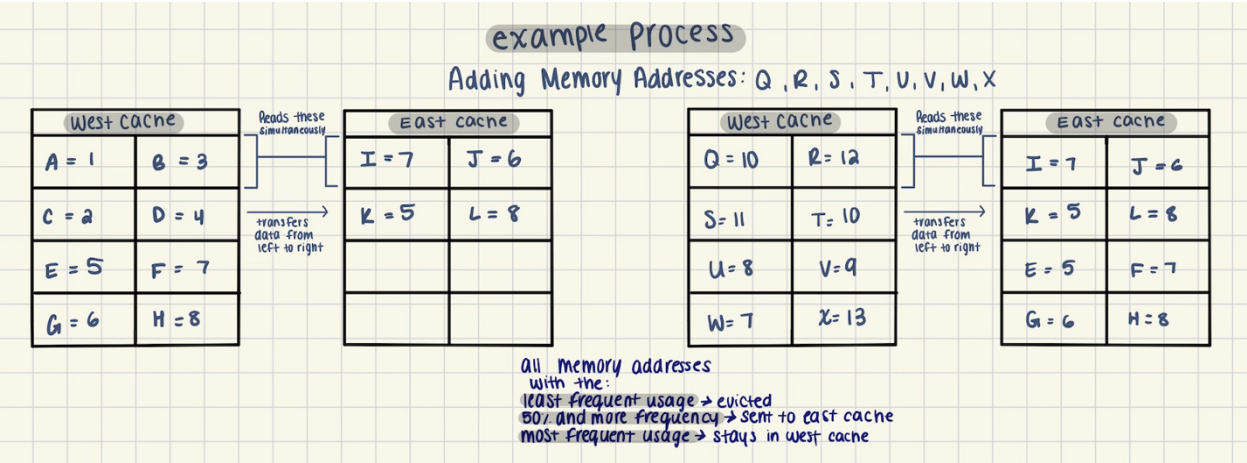


*Keep in mind that our initial desire was to use a priority queue to manage frequency. This was eventually changed to using a simple iterative algorithm, freqCheck(), that measured frequency and capacity amongst the data items.

Our unique cache strategy, the Bilateral Parallel Mapping Cache, addresses memory access efficiency issues by combining a dual-cache design with parallel searching. Unlike traditional single-cache systems, we use two parallel caches: the West Cache for recently used and new memory locations, and the East Cache for frequently used older addresses. Since both caches have the same size, each of their set indexes are the same, which allows us to search both caches adjacently. Expressing the two caches as arrays enables efficient memory management by tracking frequency rates and preserving heavily used material in the East Cache while evicting underused entries. This prioritizing improves memory use and decreases cache misses, overcoming the constraints of Least Recently Used (LRU) and Least Frequently Used (LFU) policies.

Parallel searching benefits our strategy by enabling simultaneous processing of corresponding lines from both caches, increasing hit rate and decreasing access latency. This differs from traditional set-associative caches, which loop through sets in a sequential order. We originally aimed to improve memory access efficiency, while keeping in mind the chronological pattern we discovered in the trace files. However, the longer trace files exhibited the need for temporal locality in our dual cache system because the same few addresses would be consistently accessed.

New data initially goes into an empty cache line in the set in the West cache. Then it will check if the set is full, and if it is, another data item will be replaced using our policy that evicts the oldest data item. Where that data item goes is dependent on whether it is in the top or bottom 50% frequency. If it is in the bottom 50%, it gets completely evicted from the cache system. However, if it is in the top 50%, the data item gets promoted to the East cache. In the case that it gets added to a full East cache, our strategy evicts least used data items as the replacement policy.



Implementing this new strategy presented some obstacles especially when dealing with frequency tracking counters and deciding what structure to use to do so. These challenges were resolved by continual debugging and syntactical improvements. Originally, we worked diligently

to convert the assembly code to C code before the assignment criteria changed and we were given the csim-ref.c file. Prior to this conversion process, we encountered a "no source available" error when attempting to use the “layout src” command to access the source C file. This error prevented us from analyzing the C code directly, causing confusion and delaying development of part B. After further examination, we realized that it was not possible to use this command as there was no source file, and the problem was eventually addressed when the modified assignment criteria provided the csim-ref.c file, allowing us to proceed with the project.

During the testing of our cache simulation, we also discovered a segmentation fault, which raised some doubts about how our memory management and pointer handling was implemented.

```
jb2360@ice:~/assignments/cachelab-handout$ ./csim_new -s 4 -E 1 -b 4 -t traces/yi.trace
Segmentation fault
```

We used Valgrind to analyze and troubleshoot the problem, as shown in the screenshot below.

```
jb2360@ice:~/assignments/cachelab-handout$ valgrind ./csim_new -s 4 -E 1 -b 4 -t traces/yi.trace
==3943195== Memcheck, a memory error detector
==3943195== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3943195== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3943195== Command: ./csim_new -s 4 -E 1 -b 4 -t traces/yi.trace
==3943195==
==3943195== Invalid read of size 8
==3943195==    at 0x1093F5: freqCheck (csim_new.c:105)
==3943195==    by 0x109A7F: accessData (csim_new.c:252)
==3943195==    by 0x109EC2: replayTrace (csim_new.c:327)
==3943195==    by 0x10A0AD: main (csim new.c:375)
```

Valgrind helped us locate areas of memory corruption, and after a thorough investigation, we discovered that the error occurred within a piece of the code which was handling nested loops and array accesses. We were able to fix the segmentation fault error by carefully reviewing the logic and syntactical differences, which improved the stability of our simulator and ensured reliable performance during further testing. To improve performance even more, future

additions could include taking further advantage of spatial locality and scalability to adapt to multi-level cache hierarchy.

Results and Performance Insights

```
jb2360@ice:~/assignments/cachelab-handout$ python3 test-csim_new.py

                    Your simulator               Reference simulator
Test(s,E,b)      Hits  Misses  Hit Rate    Hits  Misses  Hit Rate
0     (1,1,1):      9       8    52.9%       9       8    52.9% traces/yi2.trace
1     (4,2,4):      5       4    55.6%       4       5    44.4% traces/yi.trace
2     (2,1,4):      2       3    40.0%       2       3    40.0% traces/dave.trace
3     (1,2,2):    164      74    68.9%     128     110    53.8% traces/trans.trace
4     (2,2,3):    205      33    86.1%     201      37    84.5% traces/trans.trace
5     (3,2,4):    226      12    95.0%     226      12    95.0% traces/trans.trace
6     (5,1,5):    231       7    97.1%     231       7    97.1% traces/trans.trace
7     (5,1,5):   267573   19391    93.2%   265189   21775    92.4% traces/long.trace

Reference Total Hit Rate:     92.375%
Simulator Total Hit Rate:     93.217%


RESULT SUMMARY:
Simulator Hit Ratio Meets Criteria.
SCORE for Part B: 60/60
```

Our cache simulator outperformed the reference simulator with a total hit rate of 93.217%, demonstrating the effectiveness of our memory management and optimization strategy. Breaking down specific test cases provides additional information into our strategy's strength. For example, in test case #1 with inputs (4,2,4), our cache simulator scored a hit rate of 55.6%, which was considerably greater than the reference simulator's hit rate of 44.4%. Similarly, in test case #7 with the inputs (5,1,5), our simulator achieved a high hit rate of 93.2%, which was greater than the reference simulator's 92.4%. These results demonstrate that our technique "effectively takes advantage of temporal locality by keeping recently accessed data in the cache" while maintaining strong cache performance even under complicated data sets (Bryant & O'Hallaron, 2016, p. 535).

However, the findings suggest areas for future improvements. One example is test case #2 with inputs (2,1,4), our simulator produced a hit rate of 40.0%, which was equal to the reference simulator's performance. This indicates that our approach should be improved to effectively deal with "irregular or unpredictable access patterns, where the locality assumptions do not hold" (Bryant & O'Hallaron, 2016, p. 544). These instances, which are most likely due to insufficient implementation of spacial or temporal locality, may benefit from strategies like adaptive associativity or dynamic block size transformations. Bryant and O'Hallaron suggest that "adaptive cache replacement policies, such as least recently used (LRU) or random replacement, can be effective when access patterns are difficult to predict" (2016, p. 540). In addition, we could have delved deeper into the chronological pattern found in trace files and utilized prefetching algorithms to increase efficiency.

The use of arrays for frequency tracking ensured that frequently accessed data remained in the cache for longer periods of time, hence enabling temporal locality. Additionally, accuracy and efficiency of the accessData() function was essential in optimizing memory accesses. The higher hit rates achieved in traces like trans.trace and long.trace indicate that our architecture is successful for data that benefit from strong temporal locality. While we achieved strong results, optimizing for irregular access patterns in certain cases remains an opportunity to improve the overall performance. This combination of higher performance and adaptability highlights our strategy's potential as an important enhancement in cache design.

Group Member Contributions

Ridhima Biswas was instrumental in the development of the dual cache strategy, contributing to both the pseudocode and C implementations of the accessData() function and associated helper functions. Ridhima also helped debug and test the code to ensure its operation.

Jessica Bohn worked on converting assembly code to C for Part A of the project. She was actively involved in the design of the dual cache strategy and worked on implementing and debugging the accessData() function and overall C file.

Luke Busacca contributed to the assembly-to-C code conversion, as well as the development and implementation of the accessData() pseudocode. He developed and implemented the FreqCheck() function. His proposal for a frequency-tracking mechanism improved the cache strategy's overall performance.

Nikitha Cherukupalli worked on the dual cache strategy's design and contributed to debugging, proofreading, and troubleshooting the code. Nikitha also took the lead in producing the report, ensuring that the documentation was clear, correctly representing the project's development and the team's efforts.

Armaan Saleem worked on the associativity of the cache design and assembly-to-C conversion. He created the official diagram, which clearly explained the strategy's flow and structure. Armaan also created the hit loop for the accessData() function. He also produced the presentation slides, which helped the team effectively express the project's results.

Everyone collaborated on cache design, participated in the presentation and recitation video to effectively communicate the project results.

Reference

Bryant, R. E., & O'Hallaron, D. R. (2016). *Computer systems: A programmer's perspective* (3rd ed.). Prentice Hall.