University of Central Florida

Department of Computer Science

CDA 5106: Spring 2025

Machine Problem 1: Cache Design, Memory Hierarchy Design

Analysis of Cache Simulator

by

GROUP 11

Honor Pledge: "I have neither given nor received unauthorized aid on this test or assignment."

Student’s electronic signature: Christian Giovannetti

Student’s electronic signature: Natalie Teaman

Student’s electronic signature: Alexander Defalco

Student’s electronic signature: Patrick Rizkalla

Student’s electronic signature: Russell Ridley

Machine Problem 1: Cache Design - Analysis of Cache Simulator

1st Christian Giovanetti
Electrical and Computer Engineering
University of Central Florida
Orlando, United States of America
christian.giovannetti@ucf.edu

2nd Natalie Teaman
Electrical and Computer Engineering
University of Central Florida
Orlando, United States of America
na756834@ucf.edu

3rd Alexander Defalco
Electrical and Computer Engineering
University of Central Florida
Orlando, United States of America
alexander.defalco@ucf.edu

4th Patrick Rizkalla
Electrical and Computer Engineering
University of Central Florida
Orlando, United States of America
patrick.rizkalla@ucf.edu

5th Russell Ridley
Electrical and Computer Engineering
University of Central Florida
Orlando, United States of America
russell.ridley@ucf.edu

***Abstract*—Modern computing performance is heavily influenced by memory hierarchy design, particularly as workloads grow in complexity and scale. This work presents a flexible Java-based simulator developed to explore the performance, area, and energy trade-offs of configurable cache and memory hierarchy designs. The simulator supports single-level and two-level cache architectures under a write-back, write-allocate policy, and implements three replacement strategies: LRU, FIFO, and Optimal. It accepts configurable parameters including block size, cache size, associativity, replacement policy, and inclusion property, with support for inclusive and non-inclusive caching. Using memory traces from a subset of the SPEC-2000 benchmark suite, the simulator measures key metrics such as miss rates, average access time (AAT), and memory traffic. For optimal replacement, the simulator seeds future accesses to approximate theoretical best-case behavior. CACTI-generated models supplement simulation data with energy and area estimates. In general, this work demonstrates how varying architectural choices impact cache performance and resource efficiency, providing information on scalable and energy-sensitive memory hierarchy design.**

I. Introduction

Memory performance remains a fundamental challenge in computer architecture, particularly as modern applications demand faster data access and more efficient resource utilization. Caches serve a critical role in bridging the speed gap between CPU and main memory, reducing latency and improving overall system throughput. And as processors grow more complex, the design of cache hierarchies—including aspects like associativity, block size, and replacement policy—becomes increasingly important to optimize for performance, area, and energy. Our project focuses on the analysis of cache design using a custom-built memory hierarchy simulator. The goal is to explore how configurable cache parameters impact system behavior under different workloads. This includes evaluating the effects of inclusion policies, comparing replacement strategies such as LRU, FIFO, and Optimal, and analyzing the scalability of multi-level cache systems. Using traces from the SPEC-2000 benchmark suite and data from CACTI, we aim to identify meaningful trade-offs in cache configuration that contribute to efficient and practical memory hierarchy design.

II. Background

As computing systems continue to scale both in complexity and performance, the memory hierarchy remains one of the most important bottlenecks in modern computer architecture. Even when processor design accelerated exponentially, the disparity between CPU speeds and the time it takes to access the main memory widened, creating the requirement for an effective cache system. Caches are small, mid-level layers of quick storage that minimize memory access delay by taking advantage of the temporal and spatial locality principles. As such, effectively designed cache architectures can dramatically increase system performance, responsiveness, and power efficiency.

The project involves designing and implementing a flexible cache and memory hierarchy simulator. The ability to simulate multiple levels of cache, i.e., L1 and L2, with different configurations allowed us to see the impact of specific parameters on system behavior. They are cache size, block size, associativity, and replacement policies. One of the key motivations behind this project was to gain a deeper understanding of the trade-offs in cache design - specifically how modifying one parameter can significantly affect performance metrics such as miss rate, average access time (AAT), and memory traffic.

One of the notable features of the simulator is that it is configurable. The simulator was built to accept command-line arguments that specify a number of cache parameters such as block size, associativity, replacement policy, and inclusion property. This decision allowed us to examine a wide range of configurations from simple direct-mapped caches to fully associative designs and from single-level hierarchies to two-level inclusive caches. By doing so, we could observe and quantify how greater associativity would reduce conflict misses, or how a larger cache would reduce capacity misses.

Contextualizing our findings, we used SPEC-2000 traces that reflect realistic workload behavior. By analyzing realistic instruction streams and data access patterns, we ensured that our results would reflect true behavior rather than synthetic test cases. Moreover, the simulator was extended to support advanced replacement policies, including the Least Recently Used (LRU), First-In-First-Out (FIFO), and the Optimal policy. While the Optimal policy is not implementable in hardware due to its need for future knowledge, it serves as a theoretical baseline that allows us to compare the practical effectiveness of real-world algorithms.

The second important element of the project was the integration of CACTI-generated data for cache area and access time modeling. This enabled us to do a complete analysis that included not only performance metrics but also hardware cost metrics such as die area and energy consumption. In this way, we could compare designs that made trade-offs between low latency and low area, which is especially crucial in embedded systems and mobile computing environments where power and area budgets are small.

We also explored the inclusion property of multi-level caches i.e. the performance of inclusive vs. non-inclusive hierarchies. This aspect of the simulator allowed us to balance the trade-offs of imposing redundancy (inclusive caches) and allowing flexibility (non-inclusive caches). Inclusive caches require strict block consistency enforcement between levels, increasing complexity but possibly simplifying coherence enforcement in multiprocessor systems. Non-inclusive caches allow blocks to be present in at most one level at any given time, which can minimize redundancy but create coherence issues. Our simulator included these aspects appropriately to allow a comprehensive analysis of their implications. This project directly complemented our coursework in computer architecture, memory systems, and performance modeling. It was a chance to translate theoretical knowledge into practice on a complex, multi-faceted engineering problem. Furthermore, building a simulator from the ground up strengthened our software development skills, deepened our understanding of memory operations, and taught us how to verify system-level behavior using diff-based validation scripts.

Lastly, the project provided a valuable opportunity to bridge the gap between theoretical concepts covered in class and their practical application in real system design. By investigating every stage of the memory hierarchy, we completed it with a clearer concept of how existing systems are designed with speed, efficiency, and scalability taken into consideration. Having the ability to analyze and visualize trade-offs using plotted graphs provided us with additional visibility into cache behavior and performance tuning.

The project not only enabled us to reinforce baseline concepts in computer architecture but also positioned us well for future research or industry activity in systems design. The hands-on experience of features like optimal replacement, multi-level hierarchy management, and inclusion enforcement gave us increased appreciation for the architectural decisions that go into building real processors. It also gave us valuable practice in creating reusable, modular code skills that carry over directly to most aspects of hardware and software development.

III. Implementation & Design

To analyze the impact of configurable cache parameters on memory hierarchy performance, we implemented a trace-driven cache simulator in Java. Our simulator models both single-level (L1 only) and two-level (L1 + L2) cache hierarchies, with flexible support for a range of design parameters including block size, cache size, associativity, replacement policy, and inclusion property. The simulator runs through command-line input, where users can easily specify configuration options and provide a trace file to guide the simulation.

A. *System Architecture*

At its core, the simulator is structured around the CacheLevel class, which represents an individual cache layer (L1 or L2). Each CacheLevel contains an array of cache sets, where each set is implemented using one of three replacement policy strategies: LRU, FIFO, or Optimal. These are encapsulated in the LRUPolicy, FIFOPolicy, and OptimalPolicy classes, each inheriting from a common CacheSetBase abstract class. This design allows polymorphic behavior when selecting and applying the chosen replacement strategy.

Each cache set contains an array of CacheBlock objects, which store metadata such as tag, valid bit, dirty bit, and replacement counters. The simulator supports both write-back and write-allocate policies. On a write miss, the simulator writes data into the cache and marks the block dirty. On eviction, dirty blocks trigger writebacks to the next level or to memory, depending on cache configuration.

B. *B. Multi-Level Hierarchy and Inclusion*

If a second-level cache (L2) is specified, the simulator links the L1 and L2 caches together, such that L1 misses are passed to L2. In the case of an inclusive hierarchy, L2 evictions cause invalidations in L1 to maintain coherence. This behavior is modeled by setting parent-child references between cache levels and using invalidate() on the upper level when required.

C. *C. Optimal Replacement Support*

For simulations using the Optimal policy, we implemented a two-pass system. During the first pass, memory accesses are scanned and stored as future-access sequences for each set. These sequences allow the simulator to approximate Belady's MIN algorithm by identifying the block that will not be used for the longest time in the future. After seeding optimal data, the simulation is run as normal, with Optimal replacement logic guiding evictions.

D. *D. Trace-Driven Simulation and Statistics*

The simulator processes memory accesses from a user-supplied trace file containing read and write operations. These operations are parsed and processed one by one by the L1 cache. On misses, they are propagated to L2 if present.

Throughout execution, the simulator collects statistics including read/write counts, misses, writebacks, and total memory traffic. The Reporter class formats and prints simulation results in a standardized report, including both cache contents and performance metrics such as miss rates and memory traffic.

IV. Evaluation

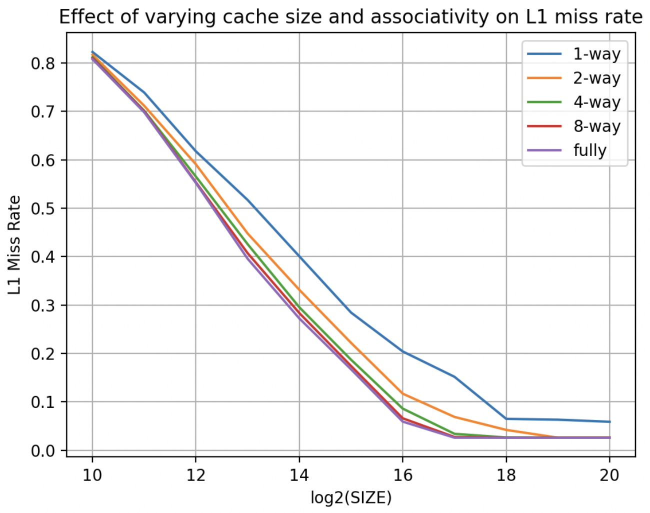

Fig. 1. Miss Rate vs. Associativity

A. Graph 1

As we observe from the graph, for any given associativity, larger caches always reduce L1 miss rate. This ought to be the case because a larger cache is able to hold more blocks and hence fewer capacity and compulsory misses. The reduction in miss rate tapers off with larger sizes of cache, suggesting diminishing returns after some size (on the order of log[base2](SIZE) = 16 or 64KB). With regard to associativity, at any size, greater associativity lowers the miss rate. This is most significant at small sizes, where conflict misses are predominant. Fully associative caches always have the lowest miss rates for all sizes since they completely eliminate conflict misses. The benefit of greater associativity diminishes at larger sizes, where the cache is so large that conflict misses are relatively infrequent. The compulsory miss rate is the floor miss rate experienced by all configurations because these misses are due to first-time accesses and cannot be avoided regardless of the cache size or associativity. The floor in the graph is observed to be around 0.025, which is the minimum value for all the associativity values for a large cache size (log[base2](SIZE) greaterthan or equal 17). To estimate the conflict miss rate for each associativity, we subtract the compulsory miss rate the low miss rate that is observed when cache size is sufficiently large from the total observed miss rate at smaller cache sizes. From the graph, the compulsory miss rate appears to level off at about 0.025, and we take this as a baseline for all of the configurations. For the direct-mapped cache (1-way associativity), this configuration has the highest miss rates for nearly all cache sizes. At log[base2](SIZE) = 10 (1KB), its miss rate is around 0.195, which, based on the implication, results in a conflict miss rate of 0.195 - 0.025 = 0.17. This high value reflects the vulnerability of the direct-mapped cache to a high conflict miss rate due to the inflexible one-block-per-set mapping. Even as cache size increases to log[base2](SIZE) = 12 (4KB), the miss rate is still 0.10, implying a conflict miss rate of 0.075, which is high. These numbers highlight the vulnerability of low-associativity caches to poor block placement when multiple addresses map to the same cache index. For the 2-way set-associative cache, the graph plots a miss rate of approximately 0.08 when log[base2](SIZE) = 12. Subtracting the compulsory component gives a conflict miss rate of 0.055. This configuration has a noticeable performance improvement over the direct-mapped cache, with a reduction in conflict misses by approximately 25 percent.For larger sizes, the 2-way curve converges more quickly to the compulsory miss rate, which indicates even small associativity significantly reduces conflict-induced evictions. Moving to the 4-way set-associative cache, we see a further reduction in conflict misses. At log[base2](SIZE) = 12, the miss rate drops to around 0.065, which corresponds to a conflict miss rate of 0.04. This improvement reflects a reducing rate of collisions due to more relaxed placement policies, with four different blocks able to share a set. The 8-way set-associative cache continues the trend. At log[base2](SIZE) = 12, it has a miss rate of about 0.055, for a conflict miss rate of 0.03. It is better than the 4-way setup, but the gains are beginning to taper. This diminishing return is to be anticipated increasing associativity reduces conflicts, but once they are mostly eliminated, additional ways have fewer marginal benefits. Finally, the fully associative cache, which theoretically eliminates all conflict misses, gives us a baseline. At small sizes like log[base2](SIZE) = 10, it has a miss rate of around 0.135, which implies a conflict miss rate of 0.11, but this is likely overstated since capacity and compulsory misses overlap at very small sizes. As the cache size is increased to log[base2](SIZE) = 12 and larger, its miss rate rapidly converges to the baseline 0.025, affirming its function to completely eradicate conflict misses. Therefore, the conflict miss rate in a fully associative cache is zero or negligible for sufficiently large sizes.

B. Graph 2

In a memory hierarchy with just an L1 cache with a block size of 32 bytes, the optimal configuration providing the lowest Average Access Time (AAT) is the fully-associative cache. From the graph, it can be seen that AAT falls with increasing cache size for all associativities, but fully-associative caches provide the optimum performance in general. This is since they minimize conflict misses, resulting in significantly reduced miss rates, a critical component in the AAT equation: AAT = Hit Time + (Miss Rate × Miss Penalty). Although the fully-associative caches carry ever so minimal higher hit latencies due to increased lookup complexity, this penalty is compen-

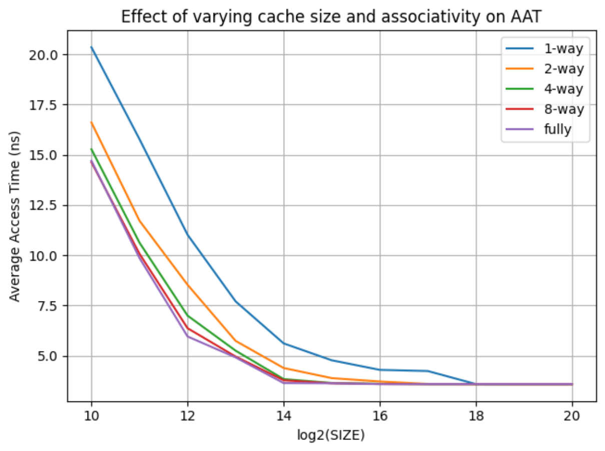

Fig. 2. Average Access Time vs. Associativity

sated for by the sharp reduction in miss rate. With the reduced cache sizes, i.e., 1KB to 4KB (log(SIZE) = 10 to 12), both 8-way and fully-associative caches are quite equally efficient. However, above a cache size of 16KB ($\log(\text{SIZE}) \geq 14$), the fully-associative configuration begins dominating all the others noticeably, eventually achieving the lowest AAT on the graph at around four nanoseconds. This makes the fully-associative cache the optimal choice for maximum access time in this system.

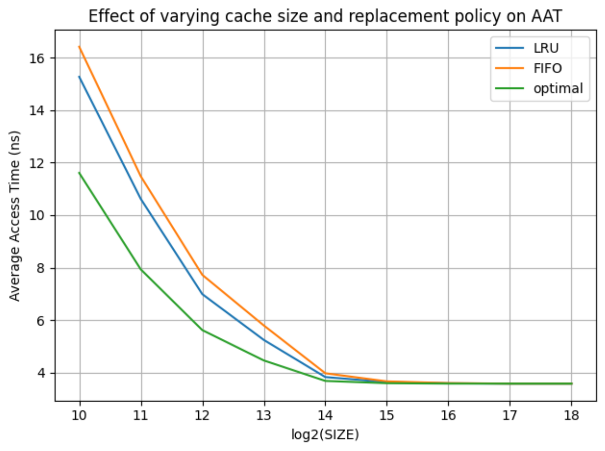

Fig. 3. Average Access Time vs. Replacement Policy

C. Graph 3

When examining how cache size and replacement policy affect Average Access Time (AAT), we observe clear trends highlighting the impact of different replacement strategies on cache performance.. For every cache size, the optimal replacement policy always has the lowest AAT. As would be expected, the ideal policy in theory evicts the block that will not be used for the longest period of time in the future, reducing misses in the cache. In smaller caches (i.e., log[base2](SIZE) = 10–12), the differences among policies are greater, and FIFO has the largest AAT, followed by LRU, and then optimal. For example, at log[base2](SIZE) = 10, the AAT for FIFO is greater than 16 ns, and LRU is slightly better at around 15 ns, while optimal falls to around 12 ns. This finding brings into focus the importance of intelligent block eviction policies, especially when memory space is limited.

For larger cache sizes, the AAT for each of the three policies converges to a similar lower bound of around 3.5 ns. This occurs because big buffers reduce total miss rate such that replacement policy choice is not so important. Even in this case, optimal policy remains a fraction ahead of LRU and FIFO. LRU is always superior since it keeps track of recently accessed blocks, which adheres more to memory access patterns (temporal locality). FIFO, however, can evict highly used blocks only because they just so happened to be put in first, and thus there are more unnecessary misses. Generally speaking, the optimal replacement policy is most effective, followed by LRU and then FIFO, especially at smaller cache sizes where eviction has more impact.

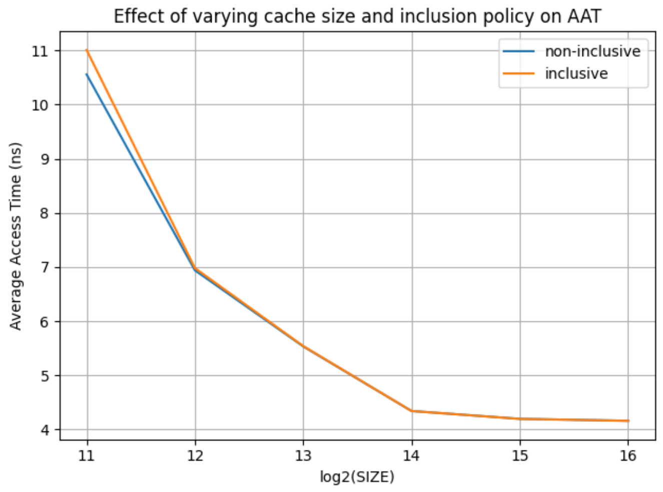

Fig. 4. Average Access Time vs. Inclusion Policy

D. Graph 4

In order to study the impact of inclusion policies on Average Access Time (AAT), we ran a series of simulations contrasting inclusive and non-inclusive cache hierarchies with various L2 cache sizes. The plot generated indicates that while both policies exhibit a clear downward trend in AAT with increasing L2 size, the non-inclusive cache always outperforms slightly. This imbalance becomes most pronounced when the L2 cache is relatively small (i.e., log(L2 SIZE) = 11, or 2KB), in which case non-inclusive caches yield an AAT of roughly 10.7 ns, compared to over 11 ns for inclusive caches.

The source of this differential performance lies fundamentally in how each of these cache types stores blocks within the memory hierarchy. In an exclusive policy, the L1 and L2 caches can possess different sets of blocks. It isolates in order to eliminate redundancy and also allows for more flexible and effective use of cached space provided. Thus, the non-inclusive caches can reside at a greater overall effective hit rate, having fewer counts of costly memory references and thereby maintaining the average time of access at a lower rate.

In contrast, inclusive caches place a restriction where any block in the L1 cache must also reside in the L2 cache. This can lead to unnecessary L2 evictions, especially under capacity pressure, which in turn can trickle down invalidations to the L1 level. This redundant residency and extra overhead can curtail the cache hierarchy's ability to cache a diverse working set, leading to increased misses and higher access times. While inclusion facilitates coherence in multi-core settings, particularly in shared cache designs, it is at the cost of reduced flexibility and efficiency in performance-tightly bound single-core or dual-core systems.

Interestingly, with an increasing size of L2 cache (naturally, when($\log(\text{SIZE}) \geq 13$), the relative performance gap between inclusive and non-inclusive policies reduces. With enough L2 capacity, both policies converge towards an AAT of around 4.2 ns. This suggests that inclusion policy impacts vanish with ample cache resources. In these situations, the working set of most programs can easily reside within the cache hierarchy, reducing the number of capacity or conflict misses under any policy employed.

Even with this convergence at larger cache sizes, non-inclusive policies always enjoy a marginal lead in all the tested cases. This consistent edge, though small, underscores the advantages of evading strict inclusion constraints, particularly in systems where cache capacity is at a premium. It also reflects a broader architectural trend toward optimizing for cache use by employing adaptive and elastic designs, of particular importance in embedded systems, mobile systems, and edge computing platforms where power and area are critical.

In short, the experiments indicate that even though inclusive caches bring ease of implementation and coherence support, particularly valuable in multi-threaded scenarios, they can introduce inefficiencies in memory-starved systems. Non-inclusive caches, through separating residency of blocks between levels, enable better use of cache and more performance in average access time. Therefore, system designers must strike a balance between the trade-offs between coherence enforcement simplicity and brute-force performance efficiency when selecting an inclusion policy, especially when designing for applications that are limited in resources or are performance-sensitive.

V. Summary

This paper presents a Java-based cache simulator designed to analyze the performance trade-offs of configurable memory hierarchies. Supporting both single- and two-level cache structures, the simulator evaluates varying block sizes, associativity levels, replacement policies (LRU, FIFO, Optimal), and inclusion properties under a write-back, write-allocate scheme. Using memory traces from the SPEC-2000 benchmark suite and CACTI-generated models, it quantifies metrics such as miss rate, average access time (AAT), memory traffic, energy consumption, and area. The findings confirm that increasing associativity and cache size generally improves performance, while optimal replacement policies and non-inclusive caching yield lower AAT—especially in constrained environments. The project bridges theoretical coursework with hands-on system design, offering insights into efficient and scalable cache architectures.

References

[1] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, no. 1, pp. 129–159, 2009. [Online]. Available: http://www.jstor.org/stable/20454196

[2] V. S. Mookerjee and Y. Tan, "Analysis of a least recently used cache management policy for web browsers," *Operations Research*, vol. 50, no. 2, pp. 345–357, 2002. [Online]. Available: http://www.jstor.org/stable/3088501