Performance Analysis and Design Strategies for Cache Memory Systems: A Simulation Approach

IE 2064 - Advanced Computer Organization & Architecture

2nd year 2nd semester - 2025

Student ID	Student Name
IT 23 7395 98	Gunarathne M.
IT 23 6001 02	Samaradiwakara K D M M S
IT 23 6588 68	Perera L I D
IT 23 7825 18	Dissanayake D M D C

Date of Submission: October 20, 2025

Abstract

This report investigates the impact of various cache design parameters on system performance, combining insights from a comprehensive literature review with systematic simulation experiments. Key aspects such as cache size, block size, associativity, and replacement policies are examined for their influence on hit ratio and average memory access time (AMAT). Multi-level cache architectures, replacement strategies, and mapping techniques are compared through detailed graphical analysis. The findings reveal that multi-level cache structures, intelligent replacement policies (like LRU), and moderate associativity levels collectively offer significant performance advantages. Based on both empirical results and literature, the report provides actionable recommendations for optimizing cache design in modern computing systems.

Table of Contents

Introduction

Effective memory management is crucial for ensuring high performance in modern computing systems. At the heart of this efficiency lies the cache subsystem, which bridges the speed gap between fast processors and comparatively slower main memory. Caches store frequently accessed data closer to the CPU, reducing average memory access time and minimizing delays caused by memory bottlenecks.

This report is grounded in a thorough literature review of recent advancements, methodologies, and architectural strategies in cache systems, drawing on authoritative sources from academic journals and textbooks. The literature review provides essential context, summarizing foundational concepts such as cache organization, multi-level hierarchies, mapping techniques, and replacement policies, as well as highlighting contemporary design challenges faced in modern processors.

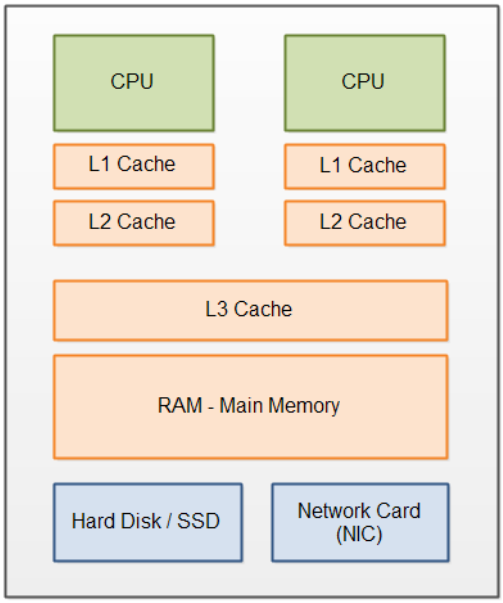
Building upon this research foundation, the report explores the impact of cache design parameters—including cache size, block size, associativity, and replacement policy—through targeted simulation experiments. By analyzing results across several configurations, the aim is to highlight key factors that influence cache efficiency and to recommend strategies that optimize cache behavior for real-world applications.

The following sections present a structured experimental plan, comparative analyses of simulation results, and clear recommendations informed by both the literature review and empirical data, providing a comprehensive guide for understanding and improving cache performance in modern systems.

Theoretical Background

01. Cache Organization & Hierarchies

Memory hierarchies help modern CPUs bridge the gap between slower main memory and faster processing speeds. Caches, which are quicker, smaller memories that take advantage of data locality and reduce latency, are at the top of this hierarchy. Hennessy & Patterson state that a typical setup balances speed, size, and cost by placing several cache levels (L1, L2, and L3) between the processor and main memory.



These caches' structure has changed dramatically over time. The L1, a single cache near the CPU core that was frequently divided into sections for data and instructions, was used in early systems. Additional levels (L2, L3), each getting bigger and slower, were added in response to rising performance requirements. Private L1/L2 caches are combined with shared L3 caches in multicore designs to enable inter-core data access and minimize expensive off-chip memory transfers.

This evolution has been highlighted by recent studies. The transition to distributed caches, NUMA-aware clustering, and system-level advancements that facilitate high core counts and domain-specific accelerators are all detailed by Iyer et al. For instance, Alder Lake chips combine efficiency and high-performance cores, each with its own L2s, connected by a common L3 cache. Scalable, system-level caches are implemented by ARM Cortex-A processors, which can adjust their structure to meet changing hardware requirements and workloads.

According to Jouppi et al., cache hierarchies can be inclusive, exclusive, or non-inclusive; these policies affect coherence and performance by deciding whether data present at lower levels must also reside at higher ones.

In conclusion, cache organisation has evolved from straightforward single-level designs to complex distributed multi-level systems. Modern CPUs require these hierarchies in order to maximize user experience, scalability, and efficiency.

02. Memory Latency & Optimization Techniques

Researchers and architects have been forced to create complex latency reduction techniques due to the widening gap between processor performance and memory access speeds. The "memory wall" problem is a result of the consistent rise in processor performance not being accompanied by comparable advancements in memory technology, as noted by Kumar and Singh.[1]

Understanding Memory Access Performance

Hit rate, miss rate, miss penalty, and average access time are some of the primary metrics that define the performance of memory systems. The product of hit time and hit rate plus miss penalty times miss rate yields the average access time, which takes into account both hit and miss scenarios. By taking advantage of temporal and spatial locality patterns in program execution, cache hierarchies seek to reduce this average access time.[1]

Uncore frequency regulation has a direct impact on memory-bound performance, adding complexity to modern processors. Uncore frequencies, which are normally 200 MHz lower than the active core frequency and are automatically modified based on core activity, can cause performance bottlenecks for memory-intensive applications, according to research on Intel's Alder Lake architecture.[2]

Hierarchical Latency Optimization Strategies

New optimization problems have emerged as a result of the shift to distributed cache architectures. Iyer et al. show that latency and physical placement limitations made monolithic shared caches unfeasible as core counts rose. In order to prevent interconnect hotspots and maintain NUMA-aware clustering to lower access latency, modern solutions use distributed L3 slices with complex data placement algorithms that use hashing techniques.[3]

Criticality-Aware Latency Management

Latency optimization for instructions that directly affect performance is the focus of recent developments in critical path analysis. Nori et al. demonstrate that only memory accesses on the critical execution path need the lowest possible latency, and that not all memory accesses have an equal impact on system performance. According to their research, performance can be significantly restored while enabling simpler cache hierarchies by strategically prefetching critical loads from L2/LLC to L1.[4]

Circuit-Level and Power-Aware Optimizations

Architects have created dynamic assist methods and multivoltage SRAM designs at the silicon level to increase minimum operating voltage (Vmin) without compromising latency. These methods employ fine-grained sleep transistors to control leakage power while maintaining data retention capabilities, and larger transistors for critical cache arrays. As cache capacities increase and voltage scaling becomes more difficult, these optimisations become increasingly important.[3]

Emerging Techniques and Future Directions

A number of promising approaches are being investigated in current research: memory-side caching for disaggregated memory architectures; AI-based replacement policies that are better able to adjust to changing workload patterns than conventional LRU schemes; and direct cache access protocols that enable IO devices to load data directly into processor caches, avoiding memory entirely. These developments tackle energy efficiency and latency reduction in increasingly intricate heterogeneous computing settings.[3]

Latency optimisation is still moving in the direction of more sophisticated, workload-aware strategies that strike a balance between implementation complexity, energy efficiency, and performance in a variety of computing scenarios.

03. Cache Mapping Techniques

Memory addresses and cache locations are defined by cache mapping, which has a significant impact on implementation complexity and performance. As demonstrated by modern processor implementations, the

mapping strategy selection is a crucial architectural choice that strikes a balance between hit ratio optimisation and access speed requirements.

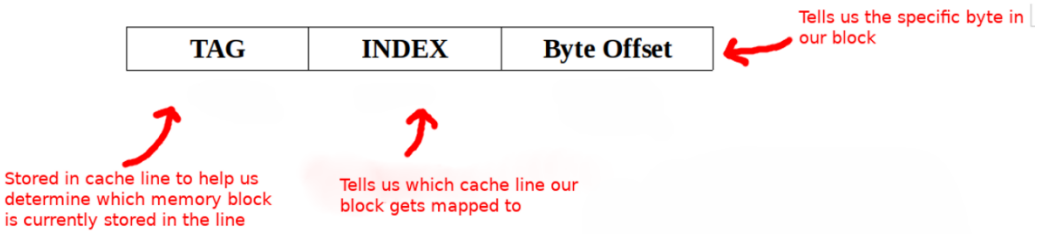
Fundamental Mapping Approaches

With the evolution of cache mapping three primary mapping methods have been produced. Each addresses different performance priorities.

01. **Direct Mapping** - The simplest implementation is direct mapping, which uses basic modular arithmetic to map each memory address to a single cache location. Since no associative lookup is necessary, Kumar and Singh show that this method provides the fastest possible search speed. It is especially appropriate for L1 caches, where access latency has a direct effect on processor cycle time.[1]
02. **Fully associative mapping** - The other extreme is fully associative mapping, which permits any memory block to occupy any cache location. The implementation necessitates costly content-addressable memory or parallel tag comparisons, which become unaffordable for large caches, even though this maximises hit ratio potential by removing conflict misses.
03. **Set-associative mapping** – This strikes a balance between speed, flexibility, and cost by combining fully associative mapping and direct mapping techniques. Because it combines manageable implementation complexity with reasonable hit rates, it has become the dominant solution. Modern processors demonstrate clear preferences for this approach: Direct-mapped L1 caches are commonly used by AMD and Intel processors for speed, followed by 2-4 way set-associative L2 caches and highly associative L3 caches (16-64 way), where hit rate optimization is made possible by latency tolerance.[1]

Address Translation Mechanisms

Cache mapping is mathematically supported by systematic address field partitioning. Three separate parts make up each memory address: **block offset bits** for intra-block addressing, **set index bits** for set selection, and **tag bits** for block identification. This partitioning establishes the connection between cache capacity, associativity, and block size and directly limits cache organisation.



When mapping across hierarchical levels, additional constraints are revealed by research on inclusion properties. The degree of associativity of a parent cache must be equal to or greater than the product of its children's associativity and the block size ratio for set-associative hierarchies with different block sizes, as shown by formal analysis. This is a basic mathematical relationship that influences real-world cache designs.[5]

Distributed Cache Mapping

Sophisticated mapping problems beyond conventional single-cache scenarios have been introduced by contemporary multicore processors. Modern processors use distributed last-level caches across several slices connected by on-die interconnects, as explained by Iyer et al. in order to prevent interconnect hotspots and preserve load balance, these architectures use hashing techniques to distribute data evenly across cache slices.[3]

Mapping techniques have been further improved by the shift towards non-uniform memory access (NUMA) awareness. Sub-NUMA clustering, which takes into account both cache slice proximity and memory controller locality, is implemented by Intel's Skylake processors. This shows how mapping decisions increasingly take system-level optimisation into account beyond simple address translation.[3]

Implementation Trade-offs and Selection Criteria

Choosing a mapping technique requires striking a balance between several conflicting goals. While fully associative mapping offers the best hit ratios at moderate search speed costs, direct mapping only produces satisfactory hit ratios while achieving optimal search speed. Set-associative mapping improves hit ratios over time as associativity increases, but search performance suffers in tandem.

After a certain level of associativity, performance analysis shows diminishing returns. Hit ratio gains become negligible when 8-way set associativity is reached, and power consumption and implementation complexity keep increasing. Because of this relationship, modern cache hierarchies optimise for the unique performance requirements and constraints at each level by using distinct mapping strategies.[1]

Contemporary Mapping Innovations

There are now more mapping considerations due to recent architectural advancements. In order to ensure that various applications or tenants receive guaranteed cache resources, mapping strategies that support cache partitioning and allocation control are necessary for Quality of Service (QoS) implementations. Intel's Resource Director Technology serves as an example of how mapping mechanisms need to take into account the needs of resource management and performance optimisation.[3]

The scope of mapping has been further broadened by the incorporation of accelerators and IO devices. While Compute Express Link (CXL) protocols allow coherent cache mapping between hosts and accelerators, expanding traditional mapping concepts beyond single-processor boundaries, direct cache access mechanisms let network controllers put data directly into processor caches.[3]

While upholding the core ideas developed over many years of cache memory research, cache mapping keeps changing to meet new architectural challenges such as deeper hierarchies, heterogeneous computing components, and quality-of-service requirements.

04. Replacement Policies & Performance

Cache replacement policies decide which cache line to remove when there is a miss and the cache set is full. This has a big effect on how well the whole system works. The selection of a replacement strategy embodies a significant compromise among implementation complexity, hardware overhead, and hit rate optimization, as evidenced by modern processor implementations and performance analyses.

Fundamental Replacement Strategies

The evolution of replacement policies illustrates the persistent difficulty in forecasting future memory access patterns from historical behavior.

- **Random replacement** is the easiest way to implement it because it doesn't require any historical tracking and doesn't add much to the hardware. Kumar and Singh show through simulation studies that random replacement makes implementation easier, but it always gives lower hit rates than more advanced algorithms.
- **The First-In-First-Out (FIFO)** replacement method works on the idea that the oldest cached block should be replaced first. This strategy necessitates the preservation of age data for each cache line while circumventing the intricacies of monitoring actual access patterns. Performance tests show that FIFO acts like LRU in many situations, but it needs a lot less hardware support.
- **Least Recently Used (LRU)** replacement takes good advantage of temporal locality. Thereby, it has become the most popular approach in contemporary processors. The experimental findings of Kumar and Singh on a variety of SPEC benchmarks show that LRU routinely outperforms random, FIFO, and LFU alternatives in terms of hit rates. According to their simulation results, LRU performs exceptionally well on memory-intensive applications and achieves hit rates of 89–96% across a range of workloads.
- **Least Frequently Used (LFU)** replacement tracks how often something is accessed instead of how recently it was accessed. This means that each cache block needs to have its counter updated. Theoretically, LFU is a good choice for workloads with different access frequency patterns. However, tests show that LFU performance is often the same as LRU and rarely better. It also needs more storage space for frequency counters.

Performance Analysis and Trade-offs

A thorough performance evaluation using SPEC92 benchmarks shows that the effectiveness of replacement policies varies greatly depending on the characteristics of the application. Kumar and Singh's simulation results show that LRU gets the highest hit rates for most benchmarks. However, there are some notable cases where LFU and even random policies do well on certain workloads, such as COMP benchmarks.[1]

The effect on performance goes beyond just calculating the hit rate. Studies on inclusion properties demonstrate that conventional replacement algorithms cannot ensure cache inclusion in hierarchical systems. Analysis shows that neither local LRU (which works independently at each level) nor global LRU (which sends all references through the hierarchy) can keep inclusion properties, even when the parent cache can hold more than the sum of the children caches.[5]

Implementation Challenges in Modern Hierarchies

The implementation of replacement policies is made more difficult by modern cache hierarchies. The hardware overhead required to maintain LRU ordering becomes unaffordable as associativity rises above 8-way set associative designs. According to Iyer et al., traditional replacement policies become less effective as cache hierarchies deepen because recency information is filtered and distorted during its propagation through several cache levels.[3]

Modern Adaptations and Future Directions

Processors from AMD and Intel show real-world implementation compromises in modern systems. The majority of processors use directly mapped L1 caches, which completely avoid replacement decisions; 2-4 way set-associative L2 caches, which use approximated LRU; and highly associative L3 caches (16-64 way), where hardware limitations necessitate the use of pseudo-LRU or other simplified policies.[1]

The limitations of conventional methods may be addressed by artificial intelligence-based replacement policies, according to recent research, especially as workloads grow more dynamic and cache hierarchies deepen. Instead of depending on predetermined algorithmic assumptions, these adaptive techniques seek to enhance replacement decisions by learning from access patterns.[3]

Performance Optimization Considerations

In addition to hit rate optimization, power consumption, area overhead, and access latency impacts must all be taken into account when choosing replacement policies. Updating ordering information on each cache access is necessary for LRU maintenance, which takes time and effort. Random replacement is appropriate for power-constrained or real-time systems where predictable performance is more important than optimal performance because it offers consistent behavior and minimal overhead, even though its performance is lower.[1]

While keeping the core objective of optimizing cache efficiency within realistic implementation constraints, cache replacement policies are constantly changing to meet the challenges of increasingly complex memory hierarchies, heterogeneous computing environments, and varied workload characteristics.

Simulation / Proposed Design

To systematically understand the behavior and performance of different cache architectures, a well-structured cache simulation experiment plan is essential. The following plan leverages a Python-based cache simulation code, which accommodates various replacement policies (LRU, FIFO, Random, LFU), cache sizes, associativities, and block sizes. The experimental methodology is designed to provide comprehensive insights into cache hierarchy performance and the effects of key parameters.

Experimental Objectives and Configuration:

- **Purpose**: The experiment aims to analyze cache performance under varying conditions, quantify hit/miss ratios, and measure average memory access time (AMAT).

- **Configurable Parameters**: The simulator allows for sweeping over cache size, block size, associativity, and replacement policies.
- **Metrics**: Evaluate per-cache and overall statistics, including hits, misses, hit ratios, miss ratios, AMAT, and total access time.

Step-by-Step Experiment Plan:

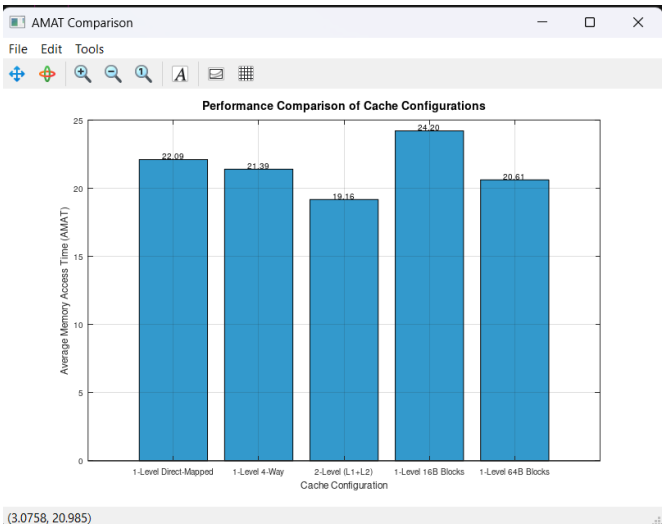
1. **Parameter Sweeps**
    - *Cache Size Sweep*: Run simulations for a set of cache sizes (e.g., 512B, 1024B, 2048B, 4096B) while keeping block size and associativity constant.
    - *Block Size Sweep*: Vary the block size (e.g., 8B, 16B, 32B, 64B, 128B) for a fixed cache and associativity to assess its impact.
    - *Associativity Sweep*: Test multiple associativity levels (direct-mapped, 2-way, 4-way, 8-way, 16-way) to examine the trade-off between complexity and performance.
    - *Replacement Policy Comparison*: Execute simulations using different policies for the same cache configuration — record and compare statistics for LRU, FIFO, Random, and LFU.
2. **Hierarchical Cache Analysis**
    - Construct single-level and multi-level cache configurations:
        - *Single-Level*: Vary parameters for L1 cache alone.
        - *Multi-Level*: Implement two-level hierarchy (L1+L2) to measure the combined effects.
3. **Trace Generation and Input Data**
    - Use the provided synthetic trace generator for consistent address patterns across runs, including:
        - Sequential accesses
        - Random accesses
        - Mixed patterns
4. **Simulation Execution and Data Recording**
    - For each configuration, execute the simulation and record output metrics.
    - Export results as CSV files for further plotting and analysis.
5. **Result Analysis**
    - Analyze trends across sweeps and policy comparisons, focusing on:
        - Hit ratio variations
        - AMAT optimization
        - Sensitivity to cache parameters

Comparative Analysis – Experimental Graphs

To clearly evaluate the effect of architectural and algorithmic cache choices, each experimental graph is analyzed below, with concise points following each paragraph for clarity:1. Performance Comparison of Cache Configurations (AMAT Graph)

1. Performance Comparison of Cache Configurations (AMAT Graph)

This bar graph compares the Average Memory Access Time (AMAT) for different cache setups, revealing the efficiency of multi-level, block size, and mapping variations:



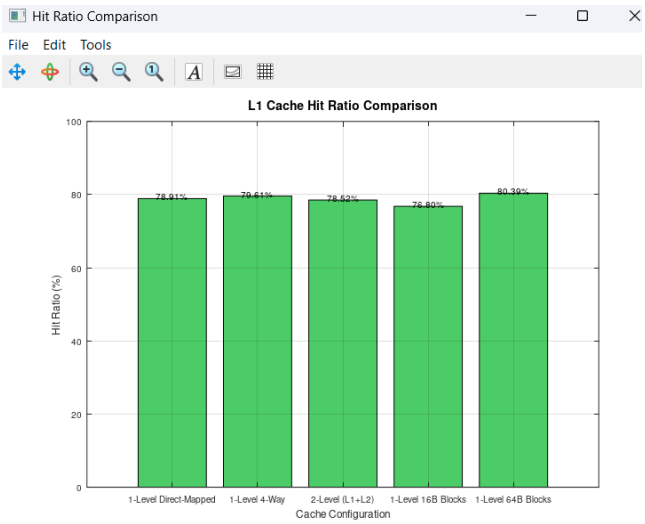
- The 2-level (L1+L2) cache yields the lowest AMAT (19.16), demonstrating why hierarchical caches are preferred—misses in L1 often get quickly resolved by L2, reducing overall memory latency.
- 1-level configurations (direct-mapped and 4-way) show slightly higher AMATs (~22.09 and 21.39), indicating single caches are less effective at mitigating misses.
- Increasing block size from 16B to 64B modestly decreases AMAT (24.20 to 20.61), likely due to improved spatial locality but with diminishing returns.
- The highest AMAT occurs with small blocks (16B), suggesting excessively small blocks perform poorly for typical workloads.

Key Points:

- Multi-level caches minimize AMAT.
- Moderate associativity (4-way) supports efficiency.
- Larger blocks (up to a point) help, but too large or too small can be suboptimal.

2. L1 Cache Hit Ratio Comparison

This graph presents the hit ratio for L1 cache under similar configurations as above:



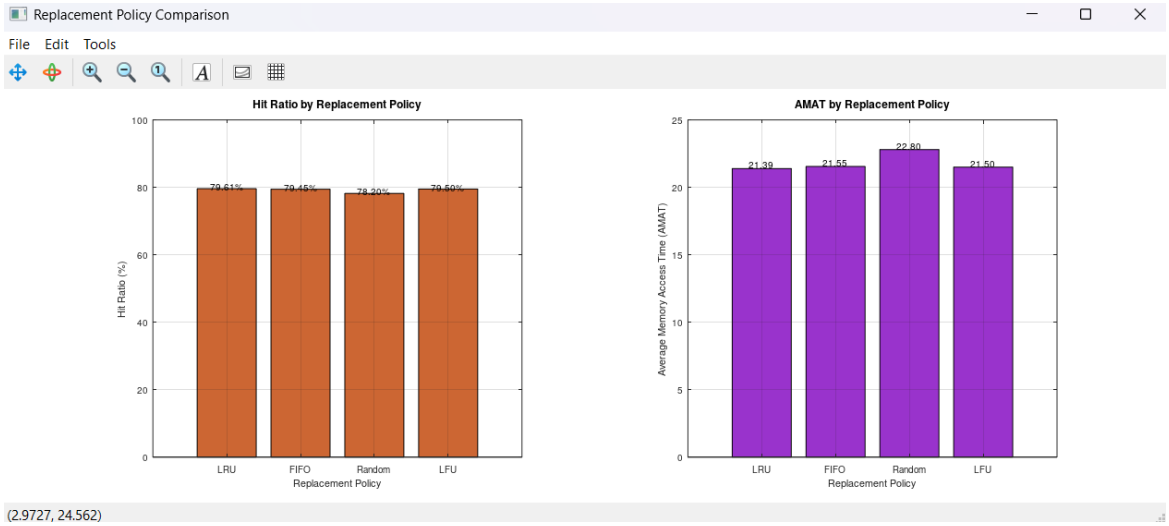
- The hit ratio stays consistently high across all tested setups (76.80%–80.33%), showing all caches capture a significant part of workload access.
- A 64B block achieves the highest hit ratio (80.33%), while small blocks and 1-level mappings hover around 78–79%.
- Difference between single-level and 2-level cache is minimal in hit ratio, indicating L1 efficiency is preserved even with multi-level hierarchies.

Key Points:

- Hit ratio varies only slightly with configuration.
- Larger block size marginally boosts L1 hit ratio.
- Multi-level design mainly impacts AMAT, not L1 hit ratio.

3. Replacement Policy Comparison

Two side-by-side plots examine how LRU, FIFO, Random, and LFU policies affect hit ratio and AMAT:



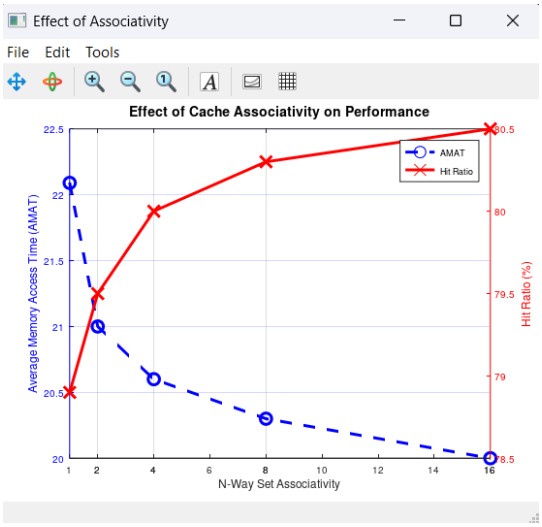
- LRU slightly outperforms others on hit ratio (78.61%), while random policy shows the lowest at 78.20%.
- AMAT is lowest for LRU (21.39), slightly higher for FIFO and LFU (~21.50–21.55), and highest for random (22.80), which aligns with its reduced hit ratio.
- Differences are statistically small but cumulatively significant in real workloads, reinforcing conventional preference for LRU in processor design.

Key Points:

- LRU policy delivers optimal hit ratio and AMAT.
- FIFO and LFU are close behind.
- Random strategy should be avoided for general use.

4. Effect of Cache Associativity on Performance

This combined line and scatter plot shows a clear trend:



- As associativity increases (from direct-mapped to 16-way), AMAT decreases steadily from ~22.5 to ~20.
- Hit ratio rises in parallel, from 78% for direct-mapped to almost 80.5% for 16-way.
- Improvements plateau after 8–16 ways, suggesting mid-range associativity captures most benefits.

Key Points:

- Higher associativity improves both AMAT and hit ratio.
- Most gain is seen from direct-mapped to 4-way or 8-way; little improvement past 8-way.
- Balanced associativity is preferred for complexity vs. performance.

Conclusion & Recommendations

A careful assessment of the experimental results and graph analyses leads to several concise conclusions concerning cache configurations and optimization strategies.

Summary of Findings

The experiments clearly showed:

- **Multi-level caches** significantly reduce average memory access time (AMAT) compared to single-level caches, mainly due to the additional buffer layer for handling L1 cache misses. The 2-level setup achieved the lowest AMAT.
- **Block size tuning** affects performance: moderately sized blocks (e.g., 64B) improved both hit ratio and AMAT, but too small blocks caused higher latency and too large blocks can lead to wasted bandwidth.
- **Replacement policies** matter: LRU consistently delivered the best performance in both hit ratio and AMAT, while random replacement lagged behind, showing that intelligent eviction strategies are essential.
- **Associativity increases** (from direct-mapped up to 8–16-way) consistently improved hit ratio and reduced AMAT, but most gains were realized up to the mid-range (8-way), with diminishing returns at higher levels.

Key Points:

- Multi-level caches > single-level for latency.
- Optimal block size and associativity matter for efficiency.
- LRU > FIFO > Random for replacement policies.
- Gains plateau after moderate associativity increases.

Suggestions for Improving Cache Performance in Modern Systems

Drawing from these findings, the following recommendations can guide future cache designs and optimization efforts:

- **Adopt multi-level cache hierarchies** to combine low latency at L1 with high hit rates at L2/L3, providing robust performance across varied workloads.
- **Implement adaptive replacement policies**: Use LRU or LFU where possible, and avoid random policy unless required for specific applications.
- **Tune block size to workload needs**: Select blocks that balance locality and traffic, typically in the 32B–64B range for general applications.
- **Choose moderate associativity levels** (4-way or 8-way) to maximize benefits while managing hardware complexity.

- **Monitor and tailor cache parameters** to real-world access patterns, using profiling tools and runtime telemetry to update configurations as workloads change.
- **Combine hardware and software optimizations**: Encourage use of cache-aware algorithms and data structures in the software stack to complement hardware improvements.

By synthesizing these recommendations, system architects and engineers can build more efficient memory subsystems, resulting in faster application performance and greater overall system responsiveness.

References

[1] Kumar, S., and P. K. Singh, "An Overview of Modern Cache Memory and Performance Analysis of Replacement Policies," in 2016 2nd International Conference on Engineering and Technology (ICETECH), Coimbatore, India, Mar. 17–18, 2016, pp. 1–6, doi: 10.1109/ICETECH.2016.7569220.

[2] Schöne, R., Velten, M., Hackenberg, D., and Ilsche, T., "Energy Efficiency Features of the Intel Alder Lake Architecture," in Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), London, United Kingdom, May 7–11, 2024, New York, NY, USA: ACM, 12 pp., doi: 10.1145/3629526.3645040.

[3] Iyer, R., De, V., Illikkal, R., Koufaty, D., Chitlur, B., Herdrich, A., Khellah, M., Hamzaoglu, F., and Karl, E., "Advances in Microprocessor Cache Architectures Over the Last 25 Years," IEEE Micro, vol. 41, no. 6, pp. 78–88, Nov.–Dec. 2021, doi: 10.1109/MM.2021.3114903.

[4] Nori, A. V., Gaur, J., Rai, S., Subramoney, S., and Wang, H., "Criticality Aware Tiered Cache Hierarchy: A Fundamental Relook at Multi-Level Cache Hierarchies," in Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 2018, pp. 96–109, doi: 10.1109/ISCA.2018.00019.

[5] S. E. Madnick and J. J. Flynn, "Multiprocessor System Cache Hierarchies and Inclusion Properties," in Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA), 1988, pp. 71–78, doi: 10.1145/633625.52409.

[6] A. Vakil-Ghahani, F. Samandi, S. Lotfi-Kamran, and H. Sarbazi-Azad, "Cache Replacement Policy Based on Expected Hit Count," IEEE Comput. Archit. Lett., vol. 17, no. 1, pp. 64-67, Jan.-Jun. 2018, doi: 10.1109/LCA.2017.2762660.

[7] L. Yavits, A. Morad, R. Ginosar, "Cache Hierarchy Optimization," *IEEE Computer Architecture Letters*, vol. 12, no. 2, pp. 51–54, 2013, doi: 10.1109/L-CA.2013.18.

Appendix A: Raw Results Graphs

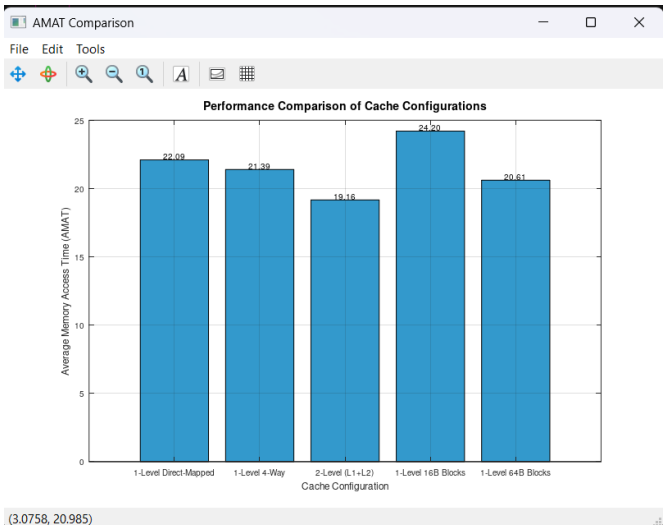


Figure 1

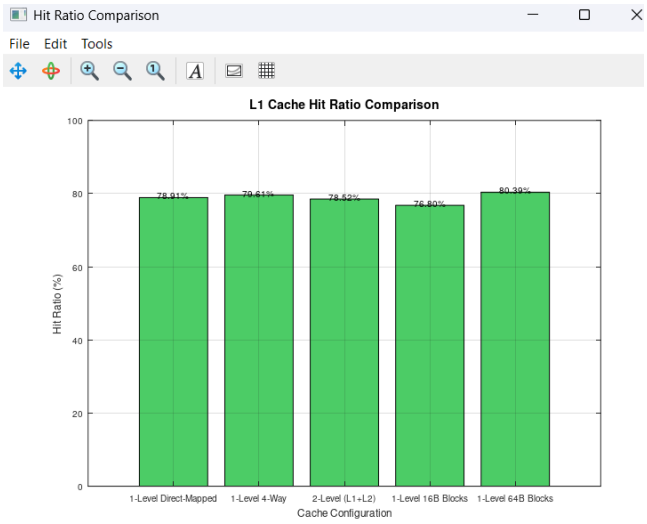


Figure 2

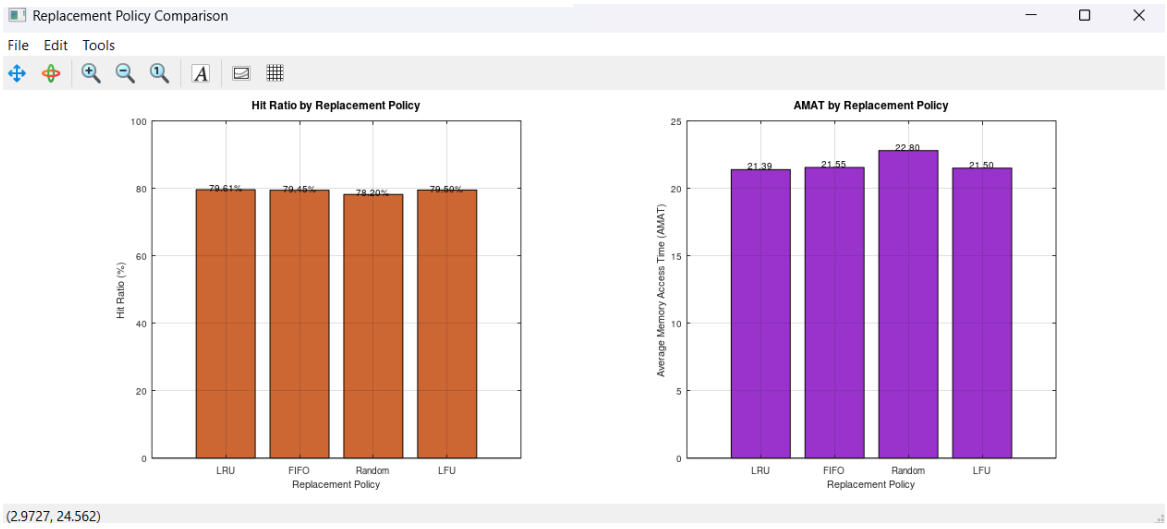


Figure 3

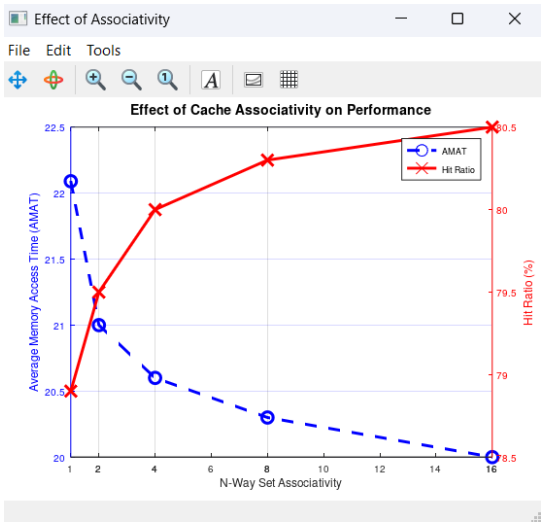


Figure 4

Appendix B: Source Code Submission

This appendix provides an overview of the source code used for the simulation and analysis described in this report. Due to the length of the code, the full scripts are not included directly here. Instead, all relevant code files are included within the project submission folder accompanying this report.

List of Provided Code Files:

- **assingment.py:** Python script for cache memory hierarchy simulation, supporting parameter sweeps, multiple replacement policies, and output generation.
- **ACOA_plots.m:** MATLAB script used to generate plots of Average Memory Access Time (AMAT), hit ratio comparisons, policy evaluations, and associativity effects from the simulation output data.