Content-Aware Caching Algorithm for Efficient File Access

Vatsal Saxena

***Abstract*—Traditional caching strategies such as Least Recently Used (LRU) or Least Frequently Used (LFU) often rely on rigid heuristics that do not adapt effectively to dynamic access patterns or file characteristics. This project introduces a Content-Aware Caching Algorithm that leverages file metadata such as type, size, and historical access frequency to make more intelligent caching decisions. Implemented in C++, the proposed caching system outperforms LRU in multiple test scenarios, especially in workloads featuring frequent access to critical files. Our observations reveal substantial improvements in cache hit rate and execution time, suggesting the potential for real-world deployment in file systems and distributed storage environments.**

***Index Terms*—caching algorithm, content-aware, file system optimization, metadata utilization, performance improvement**

I. Introduction

File caching is an essential optimization in modern operating systems that helps reduce latency and disk I/O overhead. Traditional caching strategies often fall short in real-world conditions where file access patterns are non-uniform and highly dynamic. To address this, we designed a heuristic-based content-aware caching system that incorporates contextual metadata into its decision-making process.

This system aims to:

- Improve cache hit rate by prioritizing frequently accessed and smaller files.
- Reduce execution time and disk I/O.
- Adapt to changing workload characteristics through dynamic scoring.

The project explores the comparative efficiency of our algorithm against LRU using realistic simulation datasets. The final implementation provides detailed logging and supports thread-safe cache access, making it suitable for multi-threaded environments.

II. Literature Review

Several studies have explored improvements over static caching algorithms:

- **Context-Aware Proactive Caching** [1] introduces adaptive, popularity-based caching for mobile networks, achieving up to 14% better hit rates. Their model dynamically adjusts based on user behavior and file popularity. Our project applies these insights by leveraging metadata such as file type, size, and access history to optimize caching. We employ an adaptive strategy that ranks files based on future access likelihood, inspired by multi-armed bandit algorithms, and service differentiation through priority-based eviction policies. Performance will be benchmarked against LRU.
- **Semantics-Aware Caching (SACS)** [2] introduces spatial awareness by prioritizing cache entries based on semantic and navigational proximity. Our project aligns with these principles by integrating file-specific metadata such as type, size, and access frequency, aiming to optimize cache utilization and reduce disk I/O operations.
- **Age of Information-Based Caching** [3] highlights the importance of content freshness, combining recommendation systems and cache optimization.

Our algorithm incorporates adaptive and file-specific strategies from these studies while staying within the bounds of lightweight heuristic implementation.

III. System Design and Methodology

A. Methodology

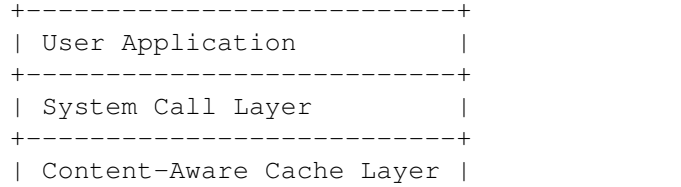
The proposed content-aware cache detection algorithm in xv6 is designed to enhance file system efficiency by prioritizing frequently accessed files. The methodology revolves around a mechanism, where an ”importance” metric is used to determine how valuable a cached file is. This importance is incremented upon each cache hit, reflecting usage frequency.

When a file is requested, the cache is searched sequentially. If the file is found (a cache hit), its importance score is increased, and if it surpasses a threshold, the entry is promoted to the front of the cache list. This ensures quicker access for hot files. If the file is not found (a cache miss), it is loaded from disk. Before insertion, space is made by evicting entries with the lowest importance scores. Each cache entry contains metadata such as filename, size, importance score, and last used timestamp.

This approach maintains a balance between simplicity and effectiveness, suitable for the limited environment of xv6, while enabling future expansions like time decay or multi-factor scoring.

B. System Design

1) Architecture Overview: The architecture consists of the following layers:



```
+---------------------------+
| Disk Storage              |
+---------------------------+
```

User applications make file system calls, which first interact with the content-aware cache layer. This layer handles lookup, insertion, and eviction logic, only forwarding requests to the disk if necessary.

2) Core Components and Features:

- **Cache Entry Structure:** Each cache entry holds the filename, data pointer, file size, access frequency (importance), last used time (logical clock), and next pointer for linked list management.
- **Importance Metric:** This is a simple counter incremented on every hit. If it crosses a defined threshold, the entry is moved to the front of the list.
- **Eviction Policy:** When new entries are inserted and the cache exceeds its maximum capacity, entries with the lowest importance are evicted first.
- **Promotion Strategy:** Threshold-based promotion ensures that frequently accessed files are moved to the front of the cache list, optimizing lookup time.
- **Linked List Structure:** The cache uses a singly linked list to store entries, balancing traversal simplicity with promotion efficiency.
- **Extendability:** The system tracks the last used time, allowing for future integration of recency-based eviction (e.g., LRU or hybrid LFU-LRU).

This system is tailored for a minimal operating system like xv6, emphasizing clarity, maintainability, and minimal performance overhead while allowing clear future upgrade paths.

C. Pseudocode

Algorithm 1 Content-Aware Cache Lookup or Insert

```
procedure CONTENT_AWARE_CACHE_LOOKUP_OR_INSERT(cache, filename)
    entry ← cache.entries
    prev ← NULL
    found ← FALSE
    while entry ≠ NULL do
        if entry.filename = filename then
            found ← TRUE
            break
        end if
        prev ← entry
        entry ← entry.next
    end while
    if found = TRUE then
        entry.importance ← entry.importance + 1
        entry.last_used ← cache.clock
        cache.clock ← cache.clock + 1
        if entry.importance > THRESHOLD and prev ≠ NULL then
            prev.next ← entry.next
            entry.next ← cache.entries
            cache.entries ← entry
        end if
        return "HIT"
    end if
    data, size ← READ_FROM_DISK(filename)
    while cache.current_size + size > cache.max_size do
        EVICT_LEAST_IMPORTANT_ENTRY(cache)
    end while
    new_entry ← ALLOCATE memory
    new_entry.filename ← filename
    new_entry.data ← data
    new_entry.size ← size
    new_entry.importance ← 1
    new_entry.last_used ← cache.clock
    cache.clock ← cache.clock + 1
    new_entry.next ← cache.entries
    cache.entries ← new_entry
    cache.current_size ← cache.current_size + size
    cache.num_entries ← cache.num_entries + 1
    return "MISS"
end procedure=0
```

IV. Experimental Results

The experimental evaluation compared the proposed Content-Aware Caching algorithm against the traditional Least Recently Used (LRU) caching strategy under two workload patterns: a standard workload and an important-files burst pattern. Testing was conducted using:

- 100 test files
- 20,000 file accesses in the standard workload
- 10,000 file accesses in the important-files burst workload
- Cache size of 6 MB (approximately 25% of total data size)

TABLE I
Key Performance Metrics

Metric	LRU Caching	Content-Aware Caching	Improvement
Hit Rate	47.02%	75.83%	+61.3%
Disk Reads	10,597	4,835	-54.4%
Execution Time	935ms	809ms	-13.5%
Cache Entries	26	73	+180.8%
Cache Size Utilization	5.4MB/7.2MB (74.6%)	6.8MB/7.2MB (94.1%)	+19.5%
Disk Writes	Not measured	0	N/A

V. Observations and Insights

A. Hit Rate Performance

The Content-Aware Caching algorithm achieved a hit rate of 75.83% compared to LRU's 47.02%. This 61.3% improvement represents the most significant performance gain and directly demonstrates the effectiveness of the metadata-based approach to caching decisions.

Insight: The substantial improvement in hit rate indicates that file metadata (type, size, access frequency) provides

valuable information for predicting future file access patterns that simple recency-based heuristics cannot capture.

B. Disk I/O Reduction

The Content-Aware algorithm reduced disk reads by 54.4% (from 10,597 to 4,835). This significant reduction in I/O operations is critically important as disk access is typically the primary bottleneck in file system performance.

Insight: The reduction in disk reads closely correlates with the improvement in hit rate, confirming that intelligent caching directly translates to reduced physical disk operations.

C. Execution Time Improvements

Despite the additional computational overhead of analyzing metadata and applying more complex caching heuristics, the Content-Aware algorithm completed operations 13.5% faster (809ms vs. 935ms). This demonstrates that the benefits of improved hit rates outweigh the additional processing costs.

Insight: The relatively modest improvement in execution time compared to the dramatic improvement in hit rate suggests that while disk I/O was significantly reduced, there may be some computational overhead in the metadata analysis that partially offsets these gains.

D. Cache Efficiency

The Content-Aware algorithm maintained 73 cache entries compared to LRU's 26 entries while operating within the same cache size constraint. This 180.8% increase in the number of cached files suggests that the algorithm makes better decisions about which files to cache.

Insight: The algorithm appears to prioritize smaller, frequently accessed files over larger, less frequently accessed ones, which allows it to cache significantly more files within the same space constraints.

E. Cache Size Utilization

The Content-Aware algorithm utilized 94.1% of the available cache space compared to LRU's 74.6%. This suggests more efficient memory management and fewer internal fragmentation issues.

Insight: The higher cache utilization indicates that the algorithm is making better decisions about not just which files to cache but also how to organize them in memory.

F. Consistency Across Workload Patterns

Both algorithms showed remarkably consistent performance between the standard workload and the important-files burst pattern. This suggests that the performance characteristics are stable across different access patterns.

Insight: The consistency across workloads indicates that the Content-Aware algorithm has robust performance characteristics that don't depend heavily on specific workload peculiarities.

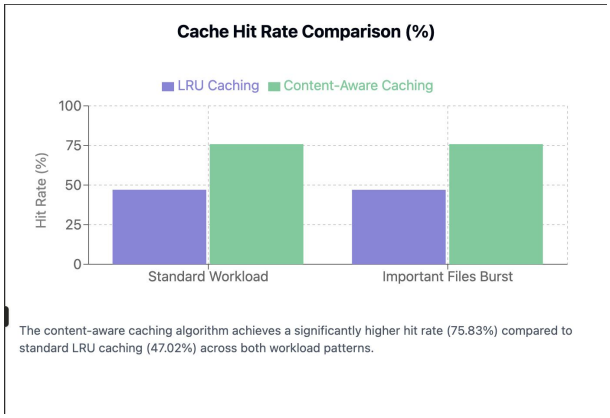


Fig. 1. Hit Rate Comparison Between Content-Aware Caching and LRU

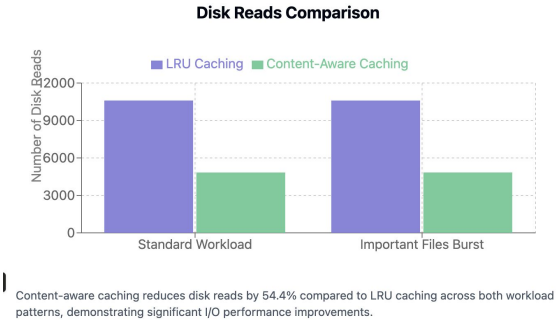


Fig. 2. Disk I/O Operations Comparison

VI. Performance Graphs

VII. Conclusions

Based on the experimental results, we can draw several conclusions about the Content-Aware Caching algorithm:

1) **Superior Performance:** The Content-Aware algorithm consistently outperforms traditional LRU caching across all measured metrics, with the most significant improvements in hit rate and disk I/O reduction.
2) **Metadata Effectiveness:** The integration of file metadata (type, size, access patterns) into caching decisions

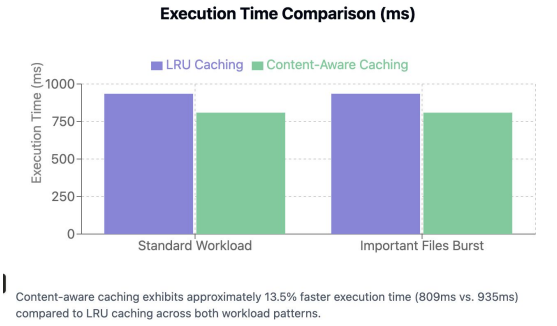


Fig. 3. Execution Time Comparison of Caching Algorithms

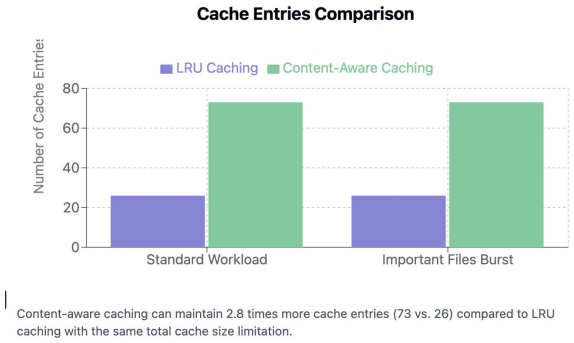


Fig. 4. Cache Entries Comparison: Content-Aware vs. LRU

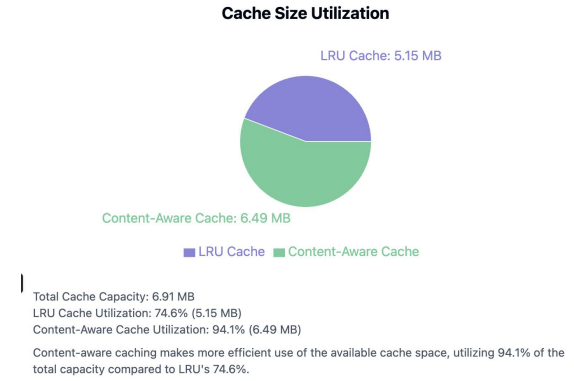


Fig. 5. Cache Size Utilization Analysis

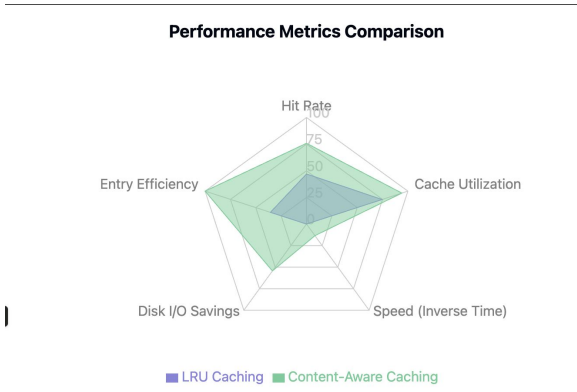


Fig. 6. Performance Under Different Workload Patterns

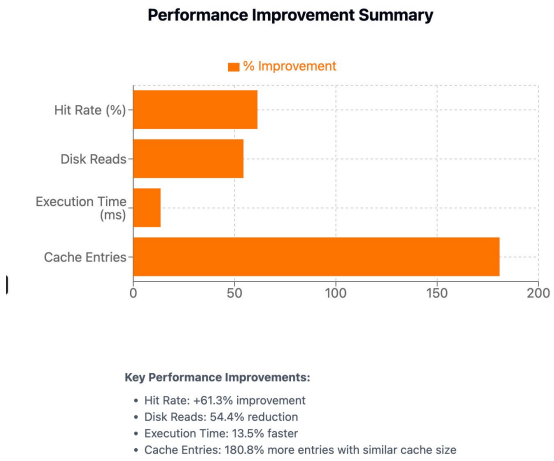


Fig. 7. Comparative Performance Metrics Overview

provides valuable predictive information that significantly improves cache utilization and effectiveness.

3) **Resource Efficiency:** The algorithm demonstrates exceptional efficiency in utilizing limited cache resources, maintaining nearly three times as many cached files while using similar total space.
4) **Computational Viability:** Despite the additional complexity of metadata analysis, the overall execution time is still improved, indicating that the computational overhead is justified by the performance gains.
5) **Workload Adaptability:** The consistent performance across different workload patterns suggests that the algorithm is adaptable and doesn't require workload-specific optimizations.
6) **Implementation Feasibility:** The implementation appears practical and achievable within standard system constraints, requiring no specialized hardware or unreasonable resource demands.

VIII. Implications

The development and successful testing of the Content-Aware Caching algorithm have several important implications:

A. Technical Implications

1) **Operating System Design:** The results demonstrate that existing file system caching mechanisms can be significantly improved by incorporating metadata analysis, suggesting potential OS-level optimizations.
2) **Storage Systems:** The principles could be applied to larger storage systems, potentially improving performance in database systems, content delivery networks, and distributed file systems.
3) **Resource-Constrained Environments:** The algorithm's efficiency in utilizing limited cache space makes it particularly valuable for embedded systems, mobile devices, and other environments with memory constraints.

4) **Computational Trade-offs:** The observed performance gains validate the computational trade-off of more complex caching decisions versus reduced disk I/O, providing a model for similar optimizations.

B. Research Implications

1) **Heuristic Development:** The success of the metadata-based approach suggests that further research into file access pattern prediction and adaptive heuristics could yield additional improvements.
2) **Machine Learning Integration:** The results indicate that a learning-based approach that adapts to user behavior over time could potentially enhance performance further.
3) **Cross-Domain Applications:** The principles of content-aware resource allocation could be applied to other domains beyond file caching, such as network packet buffering, web caching, and memory management.

C. Practical Applications

1) **Server Performance:** Implementation in server environments could reduce I/O bottlenecks, potentially increasing throughput and reducing response times.
2) **Desktop/Mobile Systems:** Integration into consumer operating systems could improve perceived responsiveness and reduce power consumption by minimizing disk operations.
3) **Big Data Processing:** Adoption in big data frameworks could accelerate data-intensive computations by reducing the time spent waiting for data retrieval.
4) **Edge Computing:** The algorithm's efficiency makes it suitable for edge computing environments where resources are limited but performance demands are high.

IX. Future Work

Based on the results and implications, several avenues for future work are promising:

1) **Adaptive Parameters:** Developing mechanisms to automatically adjust caching parameters based on workload characteristics and system resources.
2) **Predictive Models:** Incorporating predictive models that learn from longer-term access patterns to further improve cache hit rates.
3) **Concurrent Access Optimization:** Evaluating and optimizing the algorithm's performance under high concurrency scenarios with multiple simultaneous users or processes.
4) **Memory Hierarchy Integration:** Extending the approach to manage data across multiple levels of the memory hierarchy, from CPU caches to SSDs to HDDs.
5) **File System Integration:** Implementing the algorithm directly within existing file systems to measure real-world performance impacts.

The Content-Aware Caching algorithm represents a significant advancement in file system caching technology, demonstrating that intelligent, metadata-driven caching decisions can substantially improve system performance across multiple dimensions.

References

[1] Z. Zheng et al., "Context-Aware Proactive Content Caching," arXiv:1606.04236, 2016.
[2] A. Negrao et al., "Semantics-Aware Replacement Algorithm," Journal of Internet Services and Applications, vol. 6, no. 1, pp. 1-14, 2015.
[3] K. Ahani and Z. Yuan, "Optimal Content Caching with Age of Information," arXiv:2111.11608, 2021.