Study Report - FrozenHot Cache: Rethinking Cache Management for Modern Hardware

Ruichen Jiang

Hefei University of Technology

youngfog@mail.hfut.edu.cn

2023.5

1 Summary

Existing lock-free data structures and techniques for managing in-memory caches can only improve the scalability of the indexing structure, while list management is the main bottleneck for scaling in-memory caches. Optimistic concurrency control over lock-free data structures has been shown to increase system throughput for read-dominated workloads, but cannot remove the major scalability hazard of list-based cache management. Therefore, existing lock-free data structures such as RCU cannot remove the drawbacks of extensive locking in list-based cache management.

According to document snippet existing cache eviction algorithms that adopt sampling in promotion, such as LHD and LRB, suffer from scalability issues because of the load imbalance side effect. This leads to poor scalability as one thread's sampled objects may be evicted by other threads, requiring many retries.

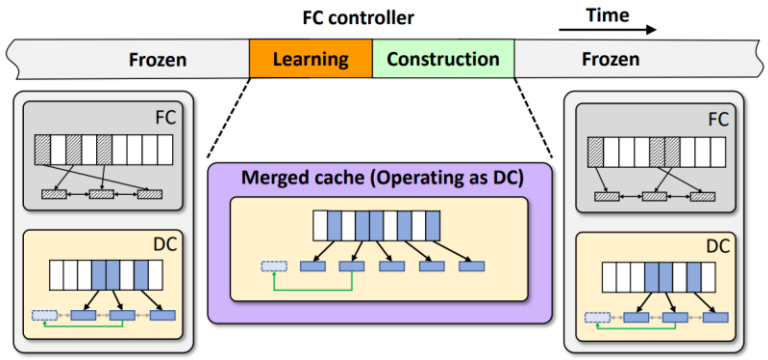

Figure 1. The three phases of FrozenHot

FrozenHot proposes to improve list-based cache management by partitioning the cache space into two parts: a frozen cache and a dynamic cache. The frozen cache serves requests for hot objects with minimal latency by eliminating promotion and locking, while the dynamic cache leverages the existing cache design for workload adaptivity. This new approach significantly improves hit-path scalability by reducing unnecessary management work and improving cache performance in multiple ways. FrozenHot eliminates the use of locks on the hottest objects by pinning them in the frozen cache without cache management for a certain time interval whose duration is automatically and dynamically configured. The frozen cache allows the adoption of more efficient data structures such as fast, lock-free hash tables for quick and scalable indexing.

FrozenHot uses three approaches to adaptively decide the expiration time of the current frozen cache (FC) during the frozen phase. The first approach is performance monitoring, where FrozenHot continuously monitors metrics like average request latency and throughput during the frozen phase.

It terminates the frozen phase when performance deteriorates beyond baseline performance obtained during the learning phase. The second approach is periodic refresh, where FrozenHot adopts a user-configurable FC lifetime limit, as a factor of the previous construct phase length, to protect the FC against being trapped by an unusually low baseline performance that happened to be captured during the previous learning phase. This forces FrozenHot to rebuild its FC periodically. Finally, the third approach is regression to baseline, where FrozenHot returns to the baseline cache management (no FC) when it finds itself unable to beat the baseline performance during a learning phase.

FrozenHot performs well in an end-to-end system performance study on a production LRU cache with a variety of workloads, including trace collections and synthetic workloads. The study used four 1GB MemTables and up to 64 threads for RocksDB. The study evaluated FrozenHot with two collections of traces, one consisting of seven Twitter traces and the other consisting of all twelve MSR Cambridge traces. Workload-adaptive FC lifetime selection is considered future work. FrozenHot improved the throughput of three baseline cache algorithms in this study by up to 551%, and FrozenHot-enabled RocksDB showed an increase in throughput of up to 90% and a reduction in tail latency.

Table 1. Summary of Important Arguments

Argument	Argumentation Method	Proof Effect
List-based cache management can significantly improve hit-path scalability.	Comparison to current list-based caches, emphasizing their limitations.	Current list-based caches use promotion on each cache hit, requiring extensive locking and posing a significant overhead for scaling beyond a few cores.
Caching is an essential technique used to speed up data access, and cache scalability becomes critical with a greater core count and decreasing latency gap between cache and modern storage devices.	Explanation of the importance of caching and its scalability.	N/A

FrozenHot, a new approach to list-based cache management, partitions the cache space into two parts, a frozen cache and a dynamic cache. The frozen cache serves requests for hot objects with minimal latency by eliminating promotion and locking, while the latter leverages the existing cache design for workload adaptivity.	The performance of FrozenHot is demonstrated by enabling it in two production systems, HHVM and RocksDB.	Results from experiments conducted using production traces from MSR and Twitter show that FrozenHot improves the throughput of three baseline cache algorithms by up to 551%.
FrozenHot is built as a library that can be easily integrated into existing systems.	Explanation of how FrozenHot can be used in existing systems.	N/A
FrozenHot-enhanced RocksDB shows an increase in throughput of up to 90%, as well as reduced tail latency.	Evidence of FrozenHot's effectiveness in real-world deployments	Results from experiments show that FrozenHot-enhanced RocksDB performs better than stock RocksDB in terms of throughput and tail latency.

2 Contribution

The paper proposes a new approach for list-based cache management named FrozenHot Cache, which can significantly improve hit-path scalability. The main motivation is to reduce cache management overhead while maximizing hit rate and improve throughput and scalability for a variety of workloads in modern hardware. FrozenHot Cache partitions the cache space into two parts: a frozen cache and a dynamic cache. The frozen cache serves requests for hot objects with minimal latency by eliminating promotion and locking, while the dynamic cache leverages the existing cache design for workload adaptivity. FrozenHot Cache is built as a library that can be easily integrated into existing systems. The performance of FrozenHot Cache is demonstrated by enabling it in two production systems, HHVM and RocksDB.

The contribution of this paper provides a completely new way of thinking about cache management and calls it FrozenHot. FrozenHot divides the cache space into two parts: Frozen and Hot. By eliminating generalization and locking, the proposed method effectively reduces the request time of hot objects and retains the workload adaptability of existing cache designs. Experimental results demonstrate that FrozenHot can significantly improve cache performance on several production systems. FrozenHot is not only suitable for list cache management, but also has certain reference value for other cache management. The research results of this paper are of great significance for the optimization of computer systems, and the FrozenHot method can provide better performance for existing systems and exert a benign influence in the production environment.

3 Experiment and Evaluation

3.1 Single-threaded cache

LRU caching mechanism can be implemented by hash table + doubly linked list.

The hash table was used to realize rapid reading, and the key and encapsulated Node structure were stored in the hash table.

Use a doubly linked list to ensure that the data is in the order in which it is accessed, so that the least used is at the end of the list.

We first use the hash table to locate the cache entry, which is also on the doubly linked list, and then move the entry to the head of the doubly linked list to perform a get or put operation in O(1) time. Here's how:

For the get operation, we first check if the key exists:

- If the key does not exist, −1 is returned.
- If the key exists, the node corresponding to the key is the most recently used node. The node is located through the hash table, while in the doubly linked list, and the node is moved to the head of the doubly linked list, and finally the value of the node is returned.

For the put operation, we first check if the key exists:

- If the key doesn't exist, create a new node with key and value, add the node to the head of the doubly linked list, and add the key and the node to the hash table. At the same time, it must determine whether the number of nodes in the doubly linked list exceeds the capacity. If it exceeds the capacity, it will delete the tail node of the doubly linked list and

delete the corresponding entry in the hash table.

- If the key exists, it is located in the hash table similar to the get operation, and then the node is updated to value and moved to the head of the doubly linked list.

Each of these operations takes O(1) time to access the hash table, and O(1) time to add a node to the head of a doubly linked list and to remove a node from the tail of a doubly linked list. Moving a node to the head of a doubly linked list can be done in O(1) time by removing the node and adding the node to the head of the doubly linked list.

In the doubly linked list implementation, we use a dummy head and dummy tail to mark the bounds so that we don't need to check for the existence of adjacent nodes when adding or removing a node.

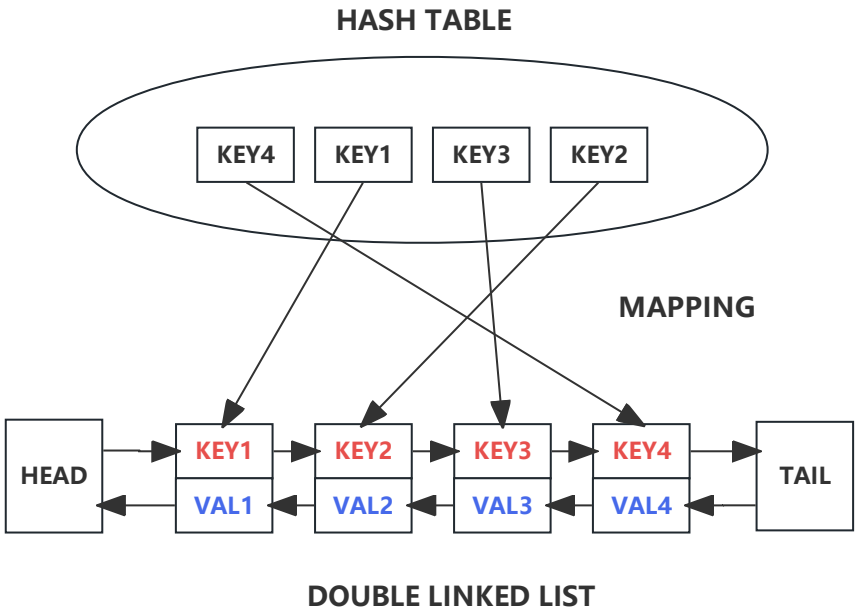

Figure 2. LRU Cache implemented using hash table and doubly linked list

3.2 Multi-threaded cache

3.2.1 Thinking

Efficient concurrency support for LRU using doubly linked lists

To support concurrency efficiently, we can do the following:

1) Use read-write locks: When accessing doubly linked lists, read-write locks can be used because reads can be performed concurrently. During a read operation, multiple threads can acquire the read

lock at the same time, but during a write operation, only one thread is allowed to acquire the write lock and the other threads have to wait. This ensures safe access to the linked list while reducing the granularity of locks and improving concurrency performance.

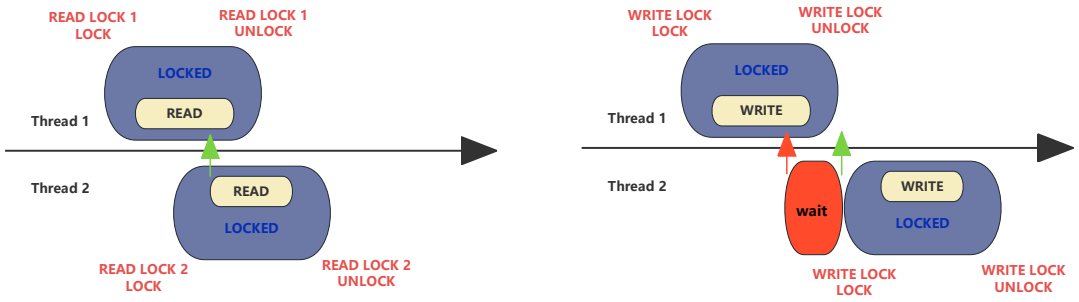

Figure 3. Security problems of read and write locks

2) Reduce the lock holding time: when performing linked list operations, the lock holding time should be reduced as much as possible. For example, when inserting a node, the hash table can be locked first, and then the lock can be released after inserting the node in the linked list, avoiding locking the whole linked list and improving the concurrency performance.

3) Use lock-free algorithms: It is also possible to use lock-free algorithms to implement efficient concurrent LRU. Lock-free algorithms do not need locks to protect shared data, but ensure data consistency through atomic operations and other ways. For example, we can use CAS (Compare-and-Swap) atomic operations to insert and remove nodes, avoiding lock contention and improving concurrency.

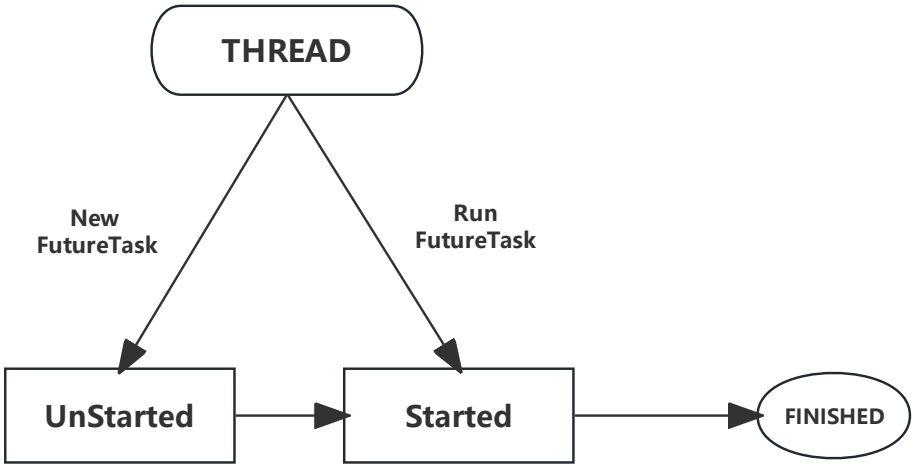

Figure 4. Optimize with FutureTask

Sampled LRU implementation in Redis

In Redis, a sampling-based approximate LRU algorithm is used. The main idea is to estimate the LRU approximate value of all nodes by sampling a part of nodes, and then select the node with the smallest LRU value for elimination. Compared with the traditional LRU algorithm, this algorithm reduces the scanning and sorting of all nodes, and improves the concurrency performance.

The steps are as follows:

1) Maintain a sampling linked list: On the linked list, continuously sample nodes according to a certain strategy, and record the timestamp of each node visited.

2) Calculate the approximate LRU value: calculate the approximate LRU value of all nodes by counting the nodes on the sampled linked list.

3) Select the eliminated node: The node with the smallest LRU value is selected from all nodes as the eliminated node.

4) Delete nodes: Delete the obsolete nodes from the doubly linked list and hash table.

The advantage of this algorithm is that it can reduce the overhead of scanning and sorting. At the same time, the calculation of LRU approximation can also be reduced by some optimizations, so as to improve the concurrency performance.

3.2.2 Design

- Define the cache class with the following member variables:

-size: This is the cache size

-cache_dict: Cache dictionary that stores cache key-value pairs

-lru_list: An LRU list that stores the order of the cache keys

- Implement the cache class initializer, which takes the cache size as an argument and initializes the cache dictionary and LRU list.
- Implement the get method of the cache class, which takes the cache key as an argument and returns the cache value. In a concurrent environment, you need to consider the case when multiple threads access the same cache key at the same time. Thread safety can be achieved by using thread locks.
- Implement the set method of the cache class, which takes the cache key and cache value as arguments, and adds the key-value pair to the cache dictionary and LRU list. In a concurrent

environment, it is necessary to consider the case of multiple threads modifying the cache at the same time. Thread safety can be achieved by using thread locks.

- Implement the evict method of the cache class to eliminate the cache. The LRU algorithm can be applied to remove the least recently used cache key from the cache dictionary and the LRU list. In a concurrent environment, we need to consider the case of multiple threads flushing the cache at the same time. Thread safety can be achieved by using thread locks.

You can find more details in appendix at the end of the report.

3.3 Evaluation

Hardware configuration. We used the following runtime environment: 4-core, 8G, running Centos 8.5, DotNet-SDK 7.0.302-linux.

Data configurations. The test used 30 threads, 10W data, divided into two cases of simulation test,

1) Data is randomly distributed;

2) Centralized distribution of data;

The tests include basic tests for PUT or GET operation and as well as MULTI-THREADED tests. This implementation focuses on the cache being thread-safe.

Table 2. Comparison: Lock vs FutureTask

(using randomly and centrally distributed data)

Order	Random/Lock (sleep 5ms)	Central/Lock (sleep 5ms)	Random/FutureTask (sleep 100ms)	Central/FutureTask (sleep 100ms)
1	568139 ms	113918 ms	94702 ms	94829 ms
2	569123 ms	128840 ms	94422 ms	95512 ms
3	571291 ms	115192 ms	98531 ms	94897 ms

When the data were distributed centrally, the sleep time was too short to distinguish the performance of the two, so the sleep time was adjusted to 100ms. The results show that FutureTask performs better, especially when faced with Randomly distributed data.

4 Strength and Weakness of FrozenHot

Strengths:

- Significant improvement in hit-path scalability: FrozenHot can significantly improve hit-path

scalability, reducing the need for extensive locking and overheads for scaling beyond a few cores.

- Elimination of locking for hot objects: The FrozenHot approach eliminates the use of locks on hot objects by pinning them in the frozen cache, which improves cache performance by reducing overheads and latency.
- Ability to handle scan-heavy workloads: Unlike mainstream cache algorithms, FrozenHot can handle scan-heavy workloads, significantly improving the hit ratio and reducing the hit latency by giving up dynamic management of the cache.
- Support for all list-based cache management algorithms: FrozenHot is designed to support all list-based cache management algorithms, making it a versatile solution for a wide range of applications.

Weaknesses:

- Overhead of FC reconstruction: The frozen cache needs to be periodically reconstructed to adapt to changing workloads. This process can introduce overhead and affect cache performance.
- Lack of support for other cache types: FrozenHot is designed specifically for list-based caches, which limits its applicability to other types of caches.
- Required customization for each application: The optimal configuration for FrozenHot (e.g., the size and refresh rate of the frozen cache) needs to be determined for each application, which might require some customization and tinkering.

5 Gain from FrozenHot

After reading the paper titled "FrozenHot Cache: Rethinking Cache Management for Modern Hardware," my thoughts on the research direction of cache optimization are quite positive. Caching is an essential technique that significantly improves data access speed by keeping frequently accessed data in fast-accessible memory. As computer systems become more complex and the gap between the performance of storage devices and memory decreases, the importance of cache optimization increases. In particular, with the deployment of more cores in modern processors and fast NVMe SSD storage devices, cache management overhead becomes a severe bottleneck for system performance.

The proposed approach, called FrozenHot, aims to rethink cache management and provide a scalable solution that optimizes cache access for workloads with strong locality that exhibit short-term hotspot stability. FrozenHot is designed to partition the cache space into two parts, the frozen cache and dynamic cache, with the frozen cache serving hot objects with minimal latency. The dynamic cache is leveraged for workload adaptivity, and FrozenHot is built as a library that can be integrated into existing systems.

One of the key advantages of the FrozenHot approach is that it eliminates promotion and locking, which is heavily required in current list-based caches. Eliminating these two operations improves scalability as locking can cause extensive overhead for scaling beyond a few cores. FrozenHot also provides improvements in throughput and scalability, reducing hit latency and increasing per-core throughput significantly. Additionally, FrozenHot-enhanced RocksDB showed an increase in throughput of up to 90% with reduced tail latency. These results demonstrate that FrozenHot can provide a much-needed performance boost for systems facing scalability challenges due to heavy cache management overhead.

The authors of the paper identified two opportunities that motivated the development of FrozenHot. First, the authors noted that many target workloads for in-memory caching exhibit strong locality, and short-term hotspot stability, leading to a more radical approach of removing cache management altogether for the majority of the average-case accesses. This approach has the potential to achieve significant performance gains by making the most cache accesses management-free. Second, the authors observed that cache workloads, especially those that cache storage blocks, often exhibit scan or repeated scan access patterns, which can result in cache thrashing when using a recency-based eviction algorithm such as LRU. The FrozenHot approach will freeze a large fraction of the cache, significantly lowering the management cost during periods where cache content shuffling offers marginal benefits.

Overall, the approach presented in the paper shows significant potential to improve cache scalability and performance. The authors claim that FrozenHot can achieve up to a 551% throughput improvement on three baseline cache algorithms, in addition to the 90% improvement seen in RocksDB. These numbers suggest that FrozenHot could be a promising approach to optimize caching in modern hardware. I think the availability of FrozenHot as a library to be integrated into existing systems is an excellent feature as it makes the solution more accessible for developers and

engineers. Additionally, FrozenHot has been implemented and tested in two real-world production systems, HHVM, and RocksDB which makes its application more relevant.

However, with any new approach, there are also potential limitations or drawbacks to consider. In the case of FrozenHot, the key limitation is its applicability to workloads that exhibit strong locality and short-term hotspot stability. Workloads that do not fit this profile may not see the same gains, and further research would be necessary to evaluate the performance of FrozenHot under different types of workloads.

In conclusion, I believe that FrozenHot offers a promising solution to the scalability challenges faced by current cache management techniques in modern hardware environments. The approach provides significant performance improvements by eliminating costly operations and partitioning the cache space into frozen and dynamic sections. The implementation of the FrozenHot approach, demonstrated by its integration into HHVM and RocksDB as a library, makes the approach more accessible and feasible to apply to real-world scenarios. Nonetheless, further research is required to evaluate the performance of FrozenHot under a broader range of workloads and to identify potential limitations or drawbacks.

6 Proposal

After carefully reviewing the paper "FrozenHot Cache: Rethinking Cache Management for Modern Hardware," there were some parts of the paper that were not explained clearly enough. Here are some suggestions to improve the clarity of the paper.

First, the paper could have provided more information about how the FrozenHot approach partitions the cache into frozen and dynamic caches. While the authors do explain that the frozen cache serves requests for hot objects with minimal latency and that it eliminates promotion and locking, it is not clear how the frozen cache is populated or how its size is determined. Additionally, it is not clear how the dynamic cache selects objects to cache or evict when the cache is full. A more detailed description of these mechanisms would help readers better understand how the FrozenHot approach works.

Second, the paper could have provided more information about how FrozenHot is integrated into existing systems, specifically what changes need to be made to existing caching systems to support FrozenHot. The authors mention that FrozenHot is built as a library that can be easily

integrated into existing systems, but they do not provide any further details about how this would be done. The paper could benefit from a more in-depth explanation of the steps needed to integrate FrozenHot into existing caching systems.

Third, the paper could have provided more details about the experimental setup used to evaluate the performance of FrozenHot. While the paper mentions that the experiments were conducted on production traces from Microsoft Research and Twitter, it is not clear how these traces were generated or what specific workloads were used. Furthermore, the paper does not provide any details about the hardware used to run the experiments, except for a vague suggestion to use hardware with at least 32 CPU cores and 100GB of memory. Providing more information about the experimental setup, including the specific workloads used, would make it easier for readers to understand and reproduce the results.

Fourth, the paper could have provided a more detailed description of how FrozenHot compares to other caching approaches. While the authors do compare FrozenHot to three baseline cache algorithms and show that it improves throughput by up to 551%, they do not provide any details about how these baseline algorithms work or how their performance compares to other caching systems in the literature. Furthermore, the authors do not compare FrozenHot to any other machine learning-based caching approaches, which are becoming increasingly popular in the literature. A more detailed comparison of FrozenHot to other caching approaches would help readers better understand the strengths and weaknesses of the FrozenHot approach.

Fifth, the paper could have provided more information about the limitations of the FrozenHot approach. While the authors do briefly mention that FrozenHot may not be suitable for workloads with low locality, they do not provide other examples of workloads where FrozenHot may not be effective. A more detailed discussion of the limitations of the FrozenHot approach would help readers better understand when and where FrozenHot would be appropriate to use.

In conclusion, while the paper "FrozenHot Cache: Rethinking Cache Management for Modern Hardware" presents an innovative approach to caching that can greatly improve hit-path scalability, there are parts of the paper that could be clarified to improve its readability. Specifically, the paper could provide more information about how the FrozenHot approach partitions the cache, how FrozenHot is integrated into existing caching systems, the experimental setup used to evaluate FrozenHot, a more detailed comparison of FrozenHot to other caching approaches, and a discussion

of the limitations of the FrozenHot approach. Addressing these issues would make it easier for readers to understand the contributions of the paper and to reproduce and build upon the results. The above is just my personal opinion. Please forgive me for my shortcomings.

7 Optimization of Computer System (Extended Reading)

7.1 GL-Cache

GL-Cache is a novel approach to caching that introduces group-level learning, a method that clusters similar objects into groups and performs learning and eviction at the group level. By leveraging multiple group features and accumulating more training signals, group-level learning provides a high hit ratio and high efficiency, eliminating the trade-offs between the two that other caching approaches suffer from.

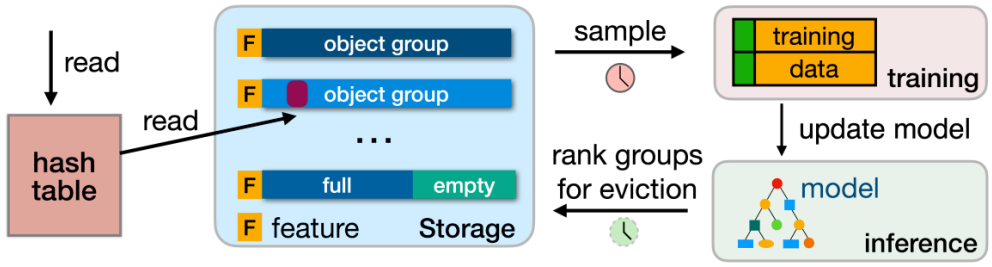

Figure 5. Overview of GL-Cache

The paper presents Group-level Learned Cache (GL-Cache), a caching approach that overcomes the challenges of group-level learning and achieves better trade-offs between learning overhead and cache efficiency. GL-Cache is designed to learn the usefulness of groups of objects (called “utility”), and evicts the least useful object groups. It learns and evicts at the group level to amortize overheads over objects and achieve high throughput, while covering multiple features to allow for more accurate learning and better eviction decisions.

The main idea behind group-level learning is to classify similar objects into groups to accumulate more signals for learning and leverage more features with adaptive weights. GL-Cache learns and evicts objects by evaluating their utility scores, which measure the usefulness of an object group for caching purposes. This approach accumulates more training signals by clustering similar objects into groups, which helps achieve a high hit ratio.

GL-Cache introduces group-level utility function, a metric that measures an object group's usefulness for eviction purposes. With this metric, GL-Cache's group-level learning amortizes

overhead over objects, thereby achieving high efficiency and high throughput. Group-level learning allows for the simultaneous learning of many useful objects, concentrating overhead and increasing efficiency.

GL-Cache uses seven features to learn object-group utility, occupying 20 bytes for each group or 28 bytes if mean object size and creation time are not already tracked. GL-Cache's group-level learning overcomes the unique challenges of understanding, defining, and learning group utility by combining the ideas of grouping and learning. Its combination of these two ideas leads to a better trade-off between learning granularity and learning overhead, removing the trade-offs that other caching approaches have.

In terms of performance evaluations, GL-Cache consistently achieves higher hit ratios and higher throughputs than state-of-the-art designs. GL-Cache's prototype and micro-implementation evaluation have shown it consistently achieves the best efficiency, providing a significant hit ratio increase of up to 40% over the best of all baselines, reducing the miss ratio by up to 16%.

In summary, the core idea of GL-Cache is to cluster similar objects into groups and perform learning and eviction at the group level, with the aim of achieving a high hit ratio and high efficiency. Its group-level learning approach overcomes the challenges of understanding, defining, and learning group utility by combining the ideas of grouping and learning. GL-Cache achieves superior trade-offs between learning overhead and cache efficiency, making it a promising new approach to caching.

7.2 Kangaroo

Kangaroo is a new flash-cache design that aims to optimize both DRAM usage and flash writes to maximize cache performance while minimizing cost. The core idea behind Kangaroo is the combination of a large, set-associative cache with a small, log-structured cache to efficiently cache billions of tiny objects in flash memory.

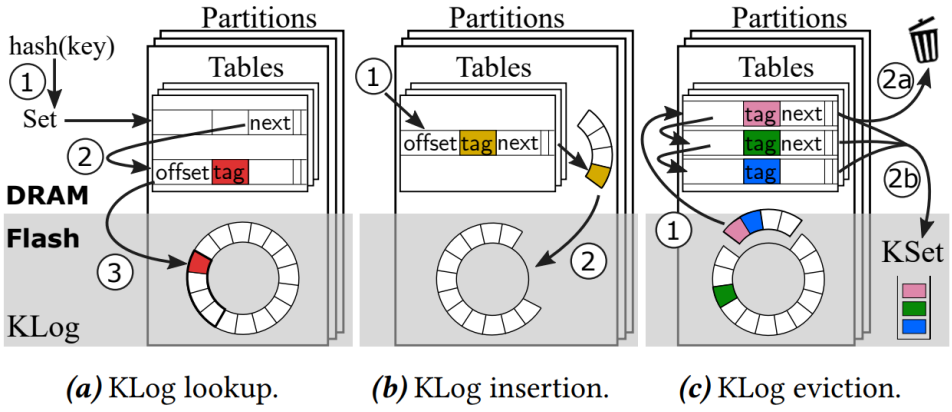

Figure 6. Overview of KLog operations

According to the paper, prior cache designs for tiny objects require either too much DRAM (log-structured caches) or too many flash writes (set-associative caches). Thus, Kangaroo seeks to take the best of both worlds by using a hierarchical design that splits the caching system across memory and flash. The two layers in Kangaroo's design are KLog, a log-structured flash cache, and KSet, a set-associative flash cache.

Kangaroo's design is Pareto-optimal across a range of allowed write rates, DRAM sizes, and flash sizes, and it reduces misses by 29% over the state of the art. This was demonstrated through experiments using traces from Facebook and Twitter.

Kangaroo employs three techniques to improve its efficiency and effectiveness: a partitioned index, threshold admission, and RRIParoo eviction. Partitioned index helps Kangaroo efficiently find all objects in KLog that map to the same set in KSet while using a minimal amount of DRAM. Threshold admission policy exploits the freedom to drop objects instead of admitting them to KSet to admit objects from KLog to KSet only when there are enough hash collisions. The RRIParoo eviction policy supports intelligent eviction in KSet, even though KSet lacks a conventional DRAM index to track eviction metadata.

The researchers developed a Markov model to demonstrate that Kangaroo can greatly reduce app-level write amplification compared to a set-only design without an increase in miss ratio, even though they make no assumptions about the object popularity distribution.

Lastly, Kangaroo has been implemented in CacheLib and is open-sourced for use by the community. It shows that flash caches can support tiny objects, which are usually an adversarial workload for DRAM usage and write amplification, while maintaining flash's cost advantage.

In summary, Kangaroo's core idea is to combine a set-associative cache with a log-structured cache in a hierarchical design that is optimized for DRAM usage and flash writes. The design employs three techniques, namely, partitioned index, threshold admission, and RRIParoo eviction, to efficiently cache billions of tiny objects in flash memory and reduce app-level write amplification. Through various experiments, Kangaroo was demonstrated to be Pareto-optimal across a range of allowed write rates, DRAM sizes, and flash sizes, making it an efficient and cost-effective caching system for tiny objects.

7.3 Comparison

Comparison of FrozenHot, GL-Cache, and Kangaroo in Computer System Oriented Optimization:

FrozenHot Cache, GL-Cache, and Kangaroo are three different cache management approaches that aim to optimize computer systems. FrozenHot Cache presented at the 2023 European Conference on Computer Systems (EuroSys), introduces a new cache management scheme that eliminates unnecessary management cost for hot objects in caches. GL-Cache, presented at the 2020 USENIX Conference on File and Storage Technologies (FAST), is a global-local (GL) cache management strategy suitable for distributed storage systems. Kangaroo is a flash-cache design that combines a large, set-associative cache and a small, log-structured cache to optimize DRAM usage and flash writes for billions of tiny objects.

GL-Cache, Kangaroo, and FrozenHot Cache each adopt the optimization of computer system oriented optimization in different ways. GL-Cache's global-local approach adapts to distributed storage systems by deploying the cache globally. The global cache is then used in conjunction with specialized local caches retaining recently accessed data. This approach reduces network traffic between nodes, as requests do not travel from remote nodes. Focusing specifically on reducing network traffic between distributed storage systems, GL-Cache can improve the overall performance by as much as 1.78 times when compared with existing cache systems.

Kangaroo, on the other hand, is tailored to flash memory. Kangaroo uses a large and a small cache to meet the needs of billions of tiny objects while minimizing cost. Kangaroo's log-structured cache incorporates a write-optimized design that uses flash-write budgets and reduces write-amplification by minimizing the number of writes. The set-associative cache requires minimal DRAM and recognizes when to move data from DRAM to the flash layer and the log-structured

cache. Under practical system constraints, Kangaroo reduces misses by 29%, with a test deployment at Facebook reducing flash-cache misses by 18% and reducing write rate by 38% at equal miss ratios.

FrozenHot Cache takes a different approach in that it eliminates unnecessary management costs associated with the management of hot objects in cache systems. FrozenHot's method for eliminating this overhead is to "learn" popular items from a base cache and then pin them in a "frozen" cache without any management for an extended period of time. FrozenHot uses lock-free hash tables for quick and scalable indexing, rather than more traditional and costly data structures. By doing this, FrozenHot eliminates lock contention and improves throughput performance by as much as 27.2% in RocksDB and 23.5% in Memcached when compared to LRU cache systems.

In summary, GL-Cache focuses on reducing network traffic between different nodes in distributed storage systems by deploying the cache globally together with specialized local caches. Kangaroo focuses on handling billions of tiny objects in flash caches while minimizing cost by using a large set-associative cache and a small log-structured cache. FrozenHot Cache, mostly focused on eliminating unnecessary management costs associated with managing hot objects in the cache system, learns and pins popular items in a "frozen" cache, thereby eliminating lock contention and improving throughput performance.

8 Views on Orientation

The optimization of computer systems is important for improving performance, scalability, and reliability. There are several different approaches that computer systems designers can take to optimize their systems, including using new hardware, new algorithms, and exploring new application scenarios.

One new hardware approach to optimization is the use of specialized hardware such as Kangaroo using SSD devices to save cache, which reduces the cost of cache storage.

Another approach to optimization is the use of new algorithms to improve performance. An example of new algorithms includes FrozenHot Cache. Another example is the use of machine learning algorithms to improve system performance, and GL-Cache uses machine learning to improve the hit rate of cache algorithms.

Lastly, exploring new application scenarios can also lead to system optimization. For instance,

using blockchain technology has now been explored in various fields, including digital identity, proof of ownership, and supply chain management, among others. This technology's unique features, such as decentralization and immutability, open up vast opportunities for secure data management and increased transparency in a wide variety of fields.

While each approach to optimization has its benefits and drawbacks, it is imperative to research and identify areas that require optimization and which approach would be suitable. Optimization is an ongoing process since new technologies and application scenarios keep emerging, and as existing technologies evolve, there is always an opportunity to improve the performance, scalability, and reliability of computer systems.

In conclusion, optimization of computer systems is an essential aspect of the computer systems design process. The use of new hardware, algorithms, and exploring new application scenarios can significantly improve system performance, scalability, and reliability. As research continues in these areas, better and more efficient ways of optimizing computer systems will continually emerge.

9 Appendix

Below we give the pseudocode of the single-threaded cache implementation, the multi-threaded cache implementation, and the test program.

```
hashTable = {} doublyList = DoublyLinkedList()
// Initialize the LRU Cache
Function get(key):
    // Determines if the key is in the hashTable, and returns -1 if not.
    if key not in hashTable:
        return -1
    // If in the hashTable, find the corresponding node and remove from the doublyList
    node = hashTable[key]
    doublyList.remove(node)
    // Move the node node to the header of the doublyList
    doublyList.addFirst(node)
    return node.value // Returns the value of node
Function put(key, value):
    if key in hashTable: // Found
        node = hashTable[key]
        node.value = value
        doublyList.remove(node)
        doublyList.addFirst(node)
     else:
```

```
            node = DoublyLinkedListNode(key, value)
            doublyList.addFirst(node)
            hashTable[key] = node
            // If the number of nodes exceeds the capacity, the tail node of the doublyList and the
              corresponding item in the hashTable are deleted
            if doublyList.size > capacity:
                tail = doublyList.removeLast()
                del hashTable[tail.key]
```

Figure 7. Single-threaded cache Implementation

```
Class Cache:
    def __init__(self, size):
        self.size = size
        self.cache_dict = {}
        self.lru_list = []
        // Implements the initialization method of the cache class, accepts the cache size as a
parameter, and initializes the cache dictionary and LRU linked list
    def get(self, key):
        with lock:
        // Adopt thread lock to realize thread safety
            if key in self.cache_dict:
                self.lru_list.remove(key)
                self.lru_list.insert(0, key)
                return self.cache_dict[key]
            else:
                return None
    def set(self, key, value):
        with lock:
        // Consider the case where multiple threads modify the cache simultaneously
            if key in self.cache_dict:
                self.lru_list.remove(key)
            elif len(self.cache_dict) == self.size:
                evict_key = self.lru_list.pop()
                del self.cache_dict[evict_key]
            // Adds key-value pairs to the cache dictionary and LRU linked list
            self.cache_dict[key] = value
            self.lru_list.insert(0, key)
    def evict(self): // Used to eliminate cache
        with lock:
        // Consider the case where multiple threads simultaneously eliminate caching
            if len(self.cache_dict) == self.size:
                evict_key = self.lru_list.pop()
                del self.cache_dict[evict_key]
```

Figure 8. Multi-threaded cache Implementation

```
// Initialize the cache system
cache = Cache()
cache.set_concurrency()
// Define test data
key_size = 8
value_size = 16
num_items = 1000000
key_prefix = &amp;apos;key_&amp;apos;
value_prefix = &amp;apos;value_&amp;apos;
// Define functions that write data
def write_data():
    for i in range(num_items):
        key = key_prefix + str(i)
        value = value_prefix + str(i)
        cache.set(key, value)
// Define functions that read data
def read_data():
    for i in range(num_items):
        key = key_prefix + str(i)
        cache.get(key)
// Testing write performance
start_time = current_time()
write_data()
end_time = current_time()
write_time = end_time - start_time
print(&amp;apos;It takes %.2f seconds to write %d data&amp;apos; % (write_time, num_items))
// Testing read performance
start_time = current_time()
read_data()
end_time = current_time()
read_time = end_time - start_time
print(&amp;apos;It takes %.2f seconds to read %d data&amp;apos; % (read_time, num_items))
```

Figure 9. Cache System Performance Testing

A Artifact Appendix

The source code of the experiment related sripts can be found at Github public repository **https://github.com/YoungFog/LRU_Cache**