Bachelor Thesis

Evaluation and Implementation of Cache Replacement Policies for an Object Store with Tired Storage

Author
`christian.grueneberg@st.ovgu.de`

October 26, 2023

First Reviewer:
Prof. Dr. Michael Kuhn

Second Reviewer:
Johannes Wünsche

Supervisor:
Johannes Wünsche

Abstract

An increasing concern is the widening disparity between processor frequency and memory latency. To address this issue, different memory technologies are combined in a memory hierarchy. Fast but expensive memory is used in combination with slower but cheaper memory to achieve an optimal balance between cost, latency, bandwidth and capacity. In addition to the memory hierarchy, caching and hierarchical storage management, also known as tiered storage, are utilized for transferring data between different hierarchy levels.

In this thesis, we implement and evaluate different cache replacement policies for a hierarchical storage stack.This storage stack is based on the B^{ε}-tree, a write-optimized variant of the B-tree. We conduct several benchmarks for different workloads, access patterns and also examine single threaded and multi threaded workloads. In particular, we are interested in how write optimization affects cache performance. Furthermore, we will give a recommendation on which cache replacement policy should be used for specific workloads.

Contents

Chapter 1.

Introduction

In this chapter, we present the motivation and research question for our thesis. We begin by examining the storage hierarchy and its resulting consequences. Next, we explore caching and hierarchical storage management, which are techniques that rely on the memory hierarchy. Using this information, we formulate our research question, which we aim to answer in this thesis. Lastly, we provide a brief outline of the structure of this paper.

1.1. Memory Hierarchy

Since the invention of integrated circuits in 1959, CPU performance has grown faster than main memory performance. Figure 1.1 shows the improvement in processor performance and DRAM performance since the 1980s. This leads to the "processor-memory performance gap", also known as the "memory wall" problem [Wulf and McKee, 1995]. One of the reasons for this is that in the semiconductor industry, CPU and memory are separate domains and have been optimized for different goals. CPUs have been optimized for higher clock frequency, whereas DRAM and hard disks have been optimized primarily for higher memory capacity [Efnusheva et al., 2017]. Even though the rates of increase in single-core performance have slowed down and thus the performance gap between processor and memory is growing more slowly, the use of multi-core processors has increased the bandwidth requirements so that the main memory must support more memory accesses.

To reduce the processor-memory performance gap, memory is built up hierarchically, as shown in Figure 1.2. The memory is ordered from fast, expensive and low capacity, which is close to the processor, to progressively slower, cheaper and higher capacity, which is further away from the processor. The goal of the memory hierarchy is to develop systems that have enough fast memory, CPU cache and DRAM to not slow down the CPU significantly and, on the other hand, have as much memory capacity as necessary provided by using hard disks, solid state disk or flash memory without excessive costs.

1.2. Caching

Closely related to the memory hierarchy is caching. Caching refers to a faster and smaller primary memory in front of a larger but slower secondary memory, so that requested data can be served faster from the primary memory instead of fetching the data from the slower memory, which improves latency and throughput and also reduces cost by using less of the

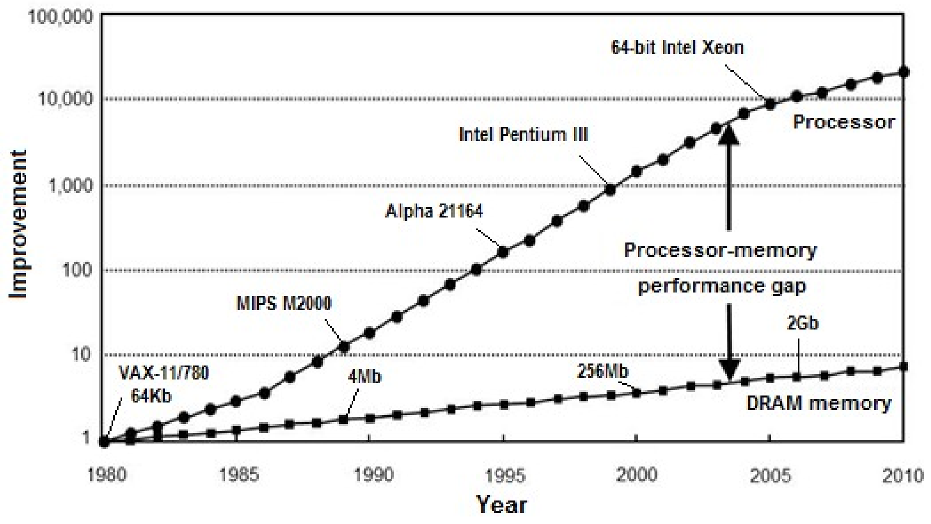


Figure 1.1.: Processor-Memory gap [Efnusheva et al., 2017]

faster memory. Caching can also mean that data from a remote memory is stored in a local memory for faster access, such as web cache. The idea of caching is based on the assumption of temporal and spatial locality. Temporal locality means that a datum accessed in the past is likely to be accessed again, and spatial locality means that when a datum is accessed, a nearby datum in memory is also likely to be accessed. To take advantage of spatial locality, blocks of data are considered instead of single datum as seen in Table 1.1.

If we include caching in the memory hierarchy so that each faster memory caches the slower memory below it, we also have a caching hierarchy as shown in Table 1.1. Caching can be done exclusively in hardware, as is the case with the CPU's L1 and L2 caches, or exclusively in software, as with the Web cache, or in a combination of both, as with the virtual memory of an operating system. Caching is an ongoing research topic, and researchers have developed various solutions for different use cases and applications.

When the cache is full and a cache miss occurs, we have to decide which cache entry should be evicted in order to move the requested block into the cache. This is one of the main problems of caches. If we displace a cache entry that we will need in the near future, we will have an additional cache miss, which we want to avoid. To decide which cache entry to remove, heuristics and algorithms, the cache replacement policies, are used.

The advantage of caching is that the latency and bandwidth, or both, are increased. But there are also drawbacks. If we use caching, we have redundant data copies in each cache which, on the one hand, means increased power consumption. Each extra copy and the movement of data consumes additional power. And on the other hand, if we have redundant data, there is an additional effort to keep the data consistent. For example, when a cache entry is modified, we need to write the data back to the original location, and we need to prevent multiple tasks from writing to a cache entry at the same time.

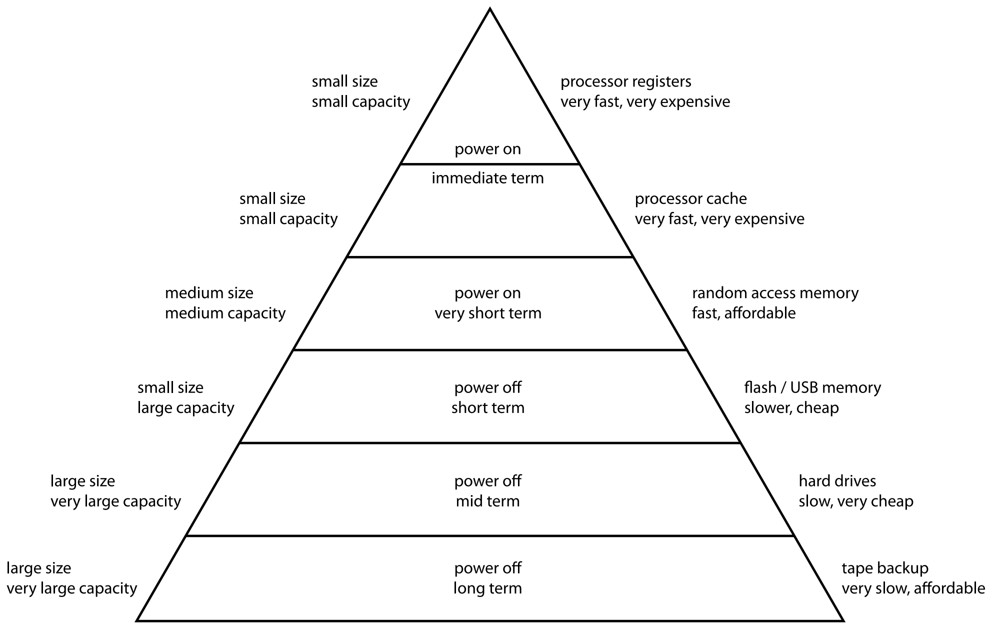


Figure 1.2.: Memory Hierarchy [Danlash, 2022]

1.3. Hierarchical Storage Management

Hierarchical storage management (HSM), also known as tiered storage, is a data management method that moves data between levels of the storage hierarchy. The data is automatically migrated between the storage media based on policies. Frequently used data is moved up the memory hierarchy to faster and more expensive storage media and infrequently used data is moved down to slower but cheaper storage media [Lugar, 2001]. Thus, HSM is a trade-off between latency and bandwidth on the one hand and cost and capacity on the other hand.

HSM has some similarities to caching. Both move data to faster storage based on policies and both have the same goal of improving I/O and reduce cost. But there are fundamental differences between them. Caching copies the data that is moved to the faster storage and use the copy instead of the original data. The original data still continues to reside in the slower storage. So, if the copy in cache is changed, the changes must be written back to the original data. HSM, on the other hand, actually moves the data to faster storage. Thus, no write back is necessary.

1.4. Summary

The growing gap between the clock frequency of the CPU on the one hand and the memory latency and bandwidth on the other hand lead to different memory technologies being built up in a memory hierarchy. This hierarchy allows a trade off between cost, capacity and access speed.

Cache Type	**What Cached**	**Where Cached**	**Latency in cycles**	**Managed By**
Registers	4-byte word	CPU registers	0	Compiler
TLB	Address translation	On-Chip TLB	0	Hardware
L1 cache	32-byte block	On-Chip L1	1	Hardware
L2 cache	32-byte block	On-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main Memory	100	Hardware + OS
Buffer Cache	Parts of files	Main Memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Table 1.1.: Caching Hierarchy [Kumar and Singh, 2016]

Yet, the question arises how the data should be moved in this hierarchy, to achieve optimal performance. Frequently used data should be located in the fast memory, while rarely used data should be moved to the slower memory. For this purpose, we have discussed two methods: caching and hierarchical storage management.

1.5. Contribution

The objective of this thesis is to implement and evaluate different cache replacement policies for a copy on write optimized hierarchical storage stack. It is of particular interest whether and how copy-on-write affects cache performance and if specific policies can be recommended for certain use cases.

1.6. Outline

We presented the context of this thesis and stated the research objectives in Chapter 1. Then, in Chapter 2, we provide an overview of topics that are important for understanding this thesis. For this purpose we will first discuss topics related to Haura, in particular B^{ε}-tree, which is the underlying data structure of Haura and the copy on write resource management technique. We then discuss the basic cache replacement policies and strategies for improving them. Afterwards, in Chapter 3, we give a brief overview of work related to the topic of this thesis. Next, in Chapter 4, we describe the newly implemented cache replacement policies and explain the changes we had to make to Haura. In Chapter 5, we design and conduct an evaluation of the newly implemented cache replacement policies. Finally, in Chapter 6, we summarize the thesis and our results in a conclusion and also suggest potential topics for future work.

Chapter 2.

Background

In this chapter we explain topics that are necessary for the understanding of the work. First, we describe the resource management technique copy on write. Then we discuss the B^ε-tree, which is a write optimized variant of the B-Tree. Next, an overview of the hierarchical storage stack Haura is given, which uses the B^ε-tree as its central data structure. We then present several basic cache replacement policies and strategies for improving them.

2.1. Copy on Write

Copy on write is a memory management technique that allows multiple processes to share data as long as the data is not modified, instead of copying the data for each process. When the data is modified, a copy is made in a different memory location and the original data remains unchanged [Ha and Kim, 2022]. Therefore, copy on write can significantly reduce the amount of memory used for shared data, but at the cost of additional overhead when the data is modified. Copy on write is used, for example, in file systems and virtual memory management. Additionally, copy on write enables the creation of snapshots, which is a read only image of a data collection that can be used as a backup or checkpoint to restore the state of a file system [Peterson, 2002]. The main limitation of copy on write is the fragmentation of memory when copies are made. This is because the new copies can be scattered across a storage medium, which increases data access time and results in increased memory usage as long as the original data and the copy are present.

2.2. B^ε-tree

The idea for the B^ε-tree was introduced by [Brodal and Fagerberg, 2003] in a study of the trade off between insertions and queries for comparison-based external memory dictionaries.

The basis for B^ε-trees is the B-tree, which has good performance on queries but suffers from poor performance on small writes, as shown in [Bender et al., 2015], because the entire node must be updated for each small write. However, if we reduce the node size to optimize for small writes, sequential read performance suffers because many smaller nodes have to be fetched from disk instead of fewer but larger ones.

To optimize for both cases, the B^ε-tree adds a buffer to each internal node of size $B - B^\varepsilon$ with $0 \leq \varepsilon < 1$ as shown in Figure 2.1.

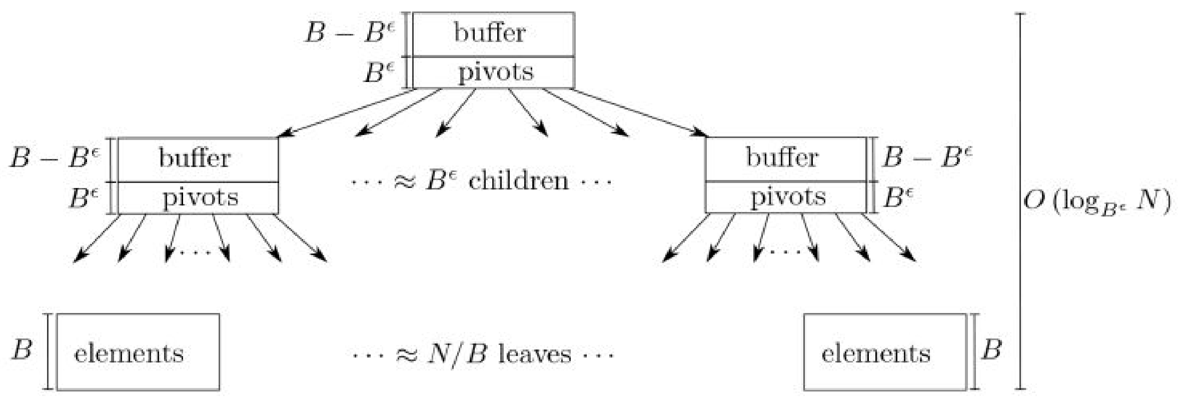

$B - B^\epsilon$
buffer
B^ϵ
pivots
$\cdots \approx B^\epsilon$ children $\cdots$
B
elements
$\cdots \approx N/B$ leaves $\cdots$
$O(\log_{B^\epsilon} N)$

Figure 2.1.: Structure B^ε-tree from [Bender et al., 2015]

As new data is inserted into the B^ε-tree, the data is written to the buffer of the root node as "insert message". Only when the root node's buffer is full, a batch of messages is flushed down the tree, preferably to the node with the most pending messages. If a node needs to be split, the buffer is also split between the new nodes. When the insert message arrives at a leaf node, the data is added to the leaf.

This behavior leads to better insertion performance than B-trees, since insertion always occurs at the root, no search for the correct insertion point is required, and the actual writing of data is delayed in batches only when enough changes have accumulated in a buffer. On deletion a "tombstone message" will be inserted into the tree and will be flushed down the tree to a leaf, like an insertion message.

In order to process queries, not only the leaf nodes must be considered as in a B-tree, additionally the buffers of the nodes must be checked to see whether they contain messages relevant for the queries. To speed up the search in a buffer for specific messages for a key or query, all buffers are organized as self-balanced search trees, like a red-black tree. Thus, the B^ε-trees achieve similar asymptotic I/O costs for queries, but are better for insertions compared to B-trees, as shown in Table 2.1.

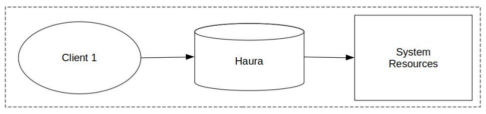
Data structure	**Insert**	**Point Query**	**Range Query**
B-tree	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
B^ε-tree	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon} + \frac{k}{B}$
B^ε-tree ($\varepsilon = 1/2$)	$\frac{\log_B N}{\sqrt{B}}$	$\log_B N$	$\log_B N + \frac{k}{B}$

Table 2.1.: The table shows the asymptotic I/O cost for B-tree and B^ε-tree. B is the node size and N the number of stored elements. The parameter ε is the ratio between the space used for storing pivots and child pointers and the space used a message buffer. Originally from [Bender et al., 2015], is shown here in shortened form.

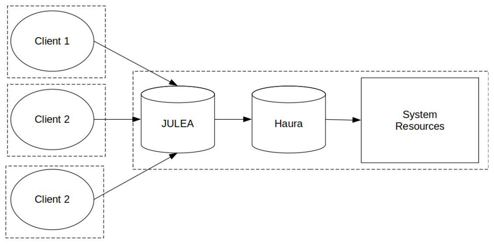
2.3. Haura

Haura is a hierarchical storage stack, which is intended as a research storage stack. It is written mainly in Rust and has the advantage that all relevant functionalities for implementing, testing and optimizing a memory stack are unified in a single code base [Wünsche, 2022]. Thus, parts can easily be modified and different approaches can be tested and evaluated against each other.

In contrast to traditional file systems, Haura runs in user space rather than kernel space. Also, Haura uses a key-value and object interface rather than the usual POSIX interface. It can be used either directly by an application, in which case Haura runs in the process context of the application, or use JULEA [Kuhn, 2017] as a wrapper. JULEA then runs the Haura instance detached from the user applications, which means that the user application can be terminated or new applications can be created and use Haura as long as JULEA is running, resulting in more flexibility compared to direct use [Wünsche, 2022].



(a) Haura direct usage.



(b) Haura usage through wrapper.

Figure 2.2.: These figures show the two usages of Haura [Wünsche, 2022].

The development of Haura started by comparing the B^{ε} tree with the ZFS and ext4 file systems [Wiedemann, 2018]. In this comparison, Haura achieved better write performance for small random writes and better sequential throughput. Additionally, Haura was extended by [Höppner, 2021] to include an object storage interface and support for multiple storage levels while retaining the benefits of the B^{ε}-tree.

Haura is structured in layers as shown in Figure 2.3. At the top is the *Database Layer* which manages multiple datasets and snapshots. Each dataset provide a key-value interface. Also each snapshots of a dataset provides a read-only key-value interface. Furthermore, for each dataset exists an own B^{ε}-tree and a root tree which saves the allocation bitmaps and metadata.

The next layer, the *Tree Layer* manages the B^{ε}-trees. The *database layer* sends messages to the B^{ε} trees consisting of key-value or key-message pairs that the B^{ε} trees process. A message for a key-message pair can contain arbitrary data or apply arbitrary code on data. Key-value pairs are stored in leave nodes and key-message nodes are stored in the inner nodes of the tree. Each node is an object for the *Data Management Layer* and is tracked individually.

The *Data Management Layer*(DML) manages objects for the *Tree Layer* and *database layer* which includes caching objects in memory, tracking modifications to objects and write back modified objects to storage devices. Main part of the DML is the Data Management Unit (DMU). The DMU is shared by all trees and ensures that no irregular state can be reached. It takes care of critical disk management such as block allocation. In addition, the DMU manages the cache, which is the main topic of this work.

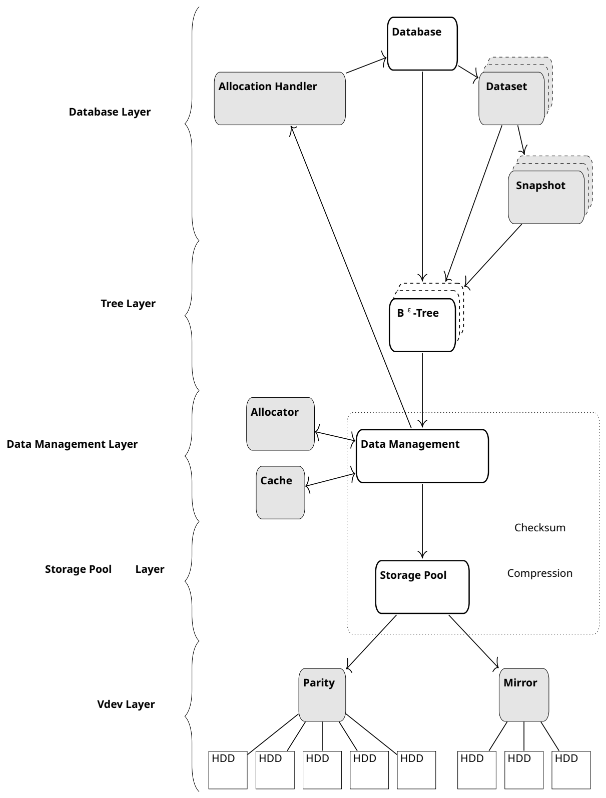


Figure 2.3.: Structure Haura [Wiedemann, 2018]

The *Storage Pool Layer* is an abstraction over the used storage hardware. Furthermore, the storage is divided into tiers from *Fastest*, *Fast*, *Slow* to *Slowest*. The user can decide how the used hardware falls into these tiers. Also, not all tiers need to be used.

The last layer, the *Vdev Layer*, is responsible for actually reading and writing data from the disks. Single disk, multiple disks or RAID-like configurations are possible.

2.4. Cache Replacement Policies

Cache replacement policies [1] are used to manage a cache. Which means that cache replacement policies decide which entries are evicted in the occurrence of a cache miss. For this decision making, they can use additional metadata or data structures. Selecting an entry for eviction is non-trivial and significantly affects the performance of the cache. Since removing an entry that will soon be accessed again results in further cache misses. To minimize these subsequent cache misses, several cache replacement strategies have been developed.

In this section, we first explain the theoretically optimal cache replacement policy, and then we discuss three early developed policies that are still widely used and also form the basis for further improved cache replacement policies.

[1]The terms "cache replacement strategies" and "cache replacement algorithm" are used interchangeably in the sources.

2.4.1. Optimal Cache Replacement Policy

The theoretically optimal cache replacement policy is Bélády's algorithm (also known as OPT or clairvoyant algorithm) [Belady, 1966]. This algorithm always evicts the cache entry whose next use is furthest in the future. The problem is that this information is usually not available during runtime, but only after an application has ended. Even though this optimal algorithm is not used in any real system, it is useful for theoretical comparison with other Cache replacement algorithm. Furthermore, many improved cache replacement strategies attempt to approximate the optimal cache replacement strategy.

2.4.2. FIFO

First in, First out (FIFO) is a simple cache replacement policy. All cache entries are stored in a queue based on a singly linked list. Each time a cache entry needs to be evicted, the entry that has been in the cache the longest is removed. This is the entry at the front of the queue, and any new entry is inserted at the back of the queue. Both operations have a constant O(1) time complexity. So, before a cache entry is evicted, it has to go through the entire queue.

The advantages of FIFO are that it is easy to implement and has a low runtime overhead. Because no further metadata needs to be saved for each entry, so on each cache hit we do not need to update the queue. This is advantageous when multiple tasks can access the cache, because we do not need locks, since neither the queue nor the metadata is altered. Furthermore, the absence of metadata and additional data structures make FIFO better suited for cases where strict size constraints must be met. We will see later that other cache replacement algorithms have ghost queues or multiple queues that can shrink and grow. So the size of a FIFO cache is predictable, or rather, there is a minimal memory overhead for FIFO compared to other replacement strategies.

However, the assumption that the entry with the longest time in the cache is always the best candidate for removal is far too simplistic and often more advanced cache replacement algorithms significantly outperform FIFO [Van Den Berg and Gandolfi, 1992]. One reason for this is that it does not distinguish between frequently or recently accessed entries that should remain in the cache, and new entries that are rarely accessed again. Every entry is treated equally. This can lead to many unnecessary evictions. Another disadvantage is that FIFO suffers from the Bélády's anomaly [Belady et al., 1969] which means that an increased cache can lead to an increased cache miss rate. The Least Recently Used replacement policy for example, does not suffer from the Bélády's anomaly.

2.4.3. Least Recently Used

Another widely used cache replacement policy is Least Recently Used(LRU). It was first mentioned in [Denning, 1980]. The idea of LRU is that whenever an entry needs to be evicted, the entry that has not been accessed for the longest time is evicted.

All cache entries are stored in a priority queue, usually implemented as doubly linked list, sorted by there last access time. Each new entry is inserted at the front of the priority queue and the entry at the back of the priority queue is always selected for eviction. In case of a cache hit, the entry that is in the priority queue is placed at the front of the queue.

Compared to FIFO, LRU performs equal or better [Van Den Berg and Gandolfi, 1992]. The reason for this is that in many cases the access distribution is skewed, so that a small number of cache entries are accessed more frequently than others. As a result of this, the entries that are frequently accessed are pushed to the top of the priority queue and thus are rarely evicted from the cache.

A disadvantage, compared to FIFO is, that we have the overhead of additional metadata, the access time or the approximation of it, for each cache entry. Thus, LRU requires more memory to store the same number of cache entries. Furthermore, cache hits also cause additional overhead. This is because a cache hit changes the priority of an entry and thus the order of priority queue changes too. Especially in cases where multiple tasks use the cache, this must happen behind a lock to ensure consistency. This can lead to additional waiting times for multiple tasks that have to wait until the cache hit has been processed.

Another disadvantage is that LRU does not use frequency information. Thus, an entry that has been accessed frequently but not recently could be evicted for an entry t hat has been accessed infrequently but recently. An example of that is looping or scanning patterns, like iterating over an array or searching for a file. Then a burst of often only once accessed entries, can lead to the eviction of entries which are accessed more frequently and should stay in the cache.

Despite these drawbacks, LRU is widely used. Partly, because of its comparatively low complexity which in turn provides good performance due to limited additional overhead. Furthermore, LRU works well for workloads with strong locality and skewed access distribution.

2.4.4. Least Frequently Used

The Least Frequently Used(LFU) policy is based on the idea that a cache entry that has been used frequently in the past will also be used in the future. Thus, when a cache miss occurs, the least frequently accessed entry is evicted. In case more then one entry has the same access count, we have to use FIFO or decide randomly which entry has to be evicted.

All cache entries are stored in a priority queue, usually implemented as doubly linked list or a min-heap, sorted by their access frequency. The entry with the lowest frequency in cache is at the back and the entry with the highest frequency in cache is at the front of the priority queue. As metadata, we have a counter assigned for each cache entry. On a cache hit, the counter for the entry will be incremented and the entry could change its position in the priority queue.

The advantage of LFU is that frequently accessed entries remain in the cache and with constant access distributions, LFU has the best cache hit ratio [Einziger et al., 2017]. But LFU has some serious drawbacks. First, the overhead for maintaining the metadata is higher than for LRU and FIFO. Also, as with LRU, there is a problem with lock contention when the priority queue is updated on a cache hit. And second, the access frequency and distribution is not constant over time. For example, if a cache entry is accessed frequently during the startup of an application but not thereafter, it may take a long time for the entry to be removed from the cache. The problem in this case is that LFU do not use recency information. There are two main strategies to make LFU more applicable in this case. The first is aging, which means that the access counter for each entry is decreased after a certain amount of time. And the second strategy is to use only a fixed time window to count accesses and after the window restart each access counter [Karakostas and Serpanos, 2000].

LFU is not used as often as LRU. Since it is more complex without performing better in most cases. However, one important area where LFU outperforms LRU is web caches. This is because web caches have a highly skewed access distribution with low locality and many entries accessed only once, [Mahanti et al., 2000].

2.5. Improved Cache Replacement Policies

All cache replacement policies presented in the last section have drawbacks. Bélády's algorithm must know the access pattern in advance and has therefore mostly theoretical value. FIFO suffers from the Bélády's anomaly and has higher miss rates than the other presented policies. LRU has problems with access patterns that have only weak locality, such as loop or scan patterns. And LFU has a high overhead and only outperforms LRU for highly skewed access patterns.

To further enhance cache performance, several additional cache replacement policies have been developed and most of these policies are based on the policies presented so far. The strategies for improvement can be divided into 4 categories:

1. Explicit user-level hints,
2. Using deeper history information,
3. Detection of access patterns,
4. Using ml-techniques,

which we will discuss in more detail below (strategies 1.-3. from [Jiang and Zhang, 2002], strategy 4. from [Rodriguez et al., 2021]).

2.5.1. Explicit User Level Hints

The first improvement strategy is through user-level hints. This idea was proposed by [Cao et al., 1994] and [Patterson et al., 1995]. The underlying concept is that the user provides hints about which cache entries have a low probability of being accessed in the near future. The problem with this approach is that the user must understand the access pattern in order to provide appropriate hints, which increases the required programming effort [Jiang and Zhang, 2002]. Therefore, user level hints can be a solution in special cases, but they are less suitable as a general cache replacement policy.

2.5.2. Utilizing Deeper History Information

The second strategy is to use more history information. LRU in particular uses very little information, only the last reference, to decide which entry to evict.

One example of this approach is LRU-K proposed by [O'neil et al., 1993]. LRU-K works similar to LRU, but instead of using only the last reference, the Kth-to-last reference is used for the eviction decision. Usually $K = 2$ is chosen, as it has shown better adaptability to different access patterns than $K > 2$. With LRU-K, cache entries that are accessed frequently have a higher chance of remaining in the cache, but entries that are rarely accessed or with a large distance between accesses, can be evicted quickly. Therefore LRU-K performs better on loops and scanning accesses patterns than a comparable LRU cache. However, LRU-K has the same major problem as LRU, which is the cost associated with using a priority queue. Any manipulation of the priority queue requires $O(\log(n)$ operations [Jiang and Zhang, 2002].

The 2Q policy of [Shasha and Johnson, 1994] attempts to achieve the same performance as LRU-K, but without the $O(\log(n)$ complexity of using a priority queue. Instead of using just one queue, 2Q uses two queues. An FIFO queue $A1$ and an LRU priority queue A_m as main cache. Also, the $A1$ is split into $A1_{in}$ for entries that actually exist in the cache and $A1_{out}$ for entries where only the reference is stored. If a cache hit occurs and the entry is in $A1_{out}$, the entry is promoted to the main cache A_m. If there is a cache hit in $A1_{in}$, the entry remains in $A1_{in}$. Also, for a cache hit in A_m, A_m behaves like a priority queue and places the entry at the beginning of A_m. The entry for eviction is either the front entry of the $A1_{in}$ queue, if the length of the $A1_{in}$ is greater then a certain threshold, or the tail of the A_m otherwise. Like LRU-K, 2Q is better suited for looping and scanning access patterns. This is because only once referenced entries leave the cache quickly. However, the advantage is that 2Q has a constant time overhead. The disadvantage, on the other hand, is that the length of the $A1_{in}$ and $A1_{out}$ queues is predefined. These need to be tuned for each use case to achieve optimal performance. This limits the use of 2Q as a general cache replacement policy.

Another example is Least Recently/Frequently Used(LRFU) policy [Lee et al., 2001]. LRFU combines LRU and LFU methods by tracking the frequency and reference for each entry and using a weighting factor λ to decide which is more important. The problem with this approach is that the λ is fixed. It is therefore not adaptable to changing access patterns. Furthermore, the λ parameter is highly dependent on the hardware used and the access pattern. We must therefore determine the correct λ for each use case. Therefore, LRFU is also not suitable as a general cache replacement algorithm.

2.5.3. Detection and Adaption of Access Patterns

The third strategy is to identify specific access patterns in the history information for certain entries, based on either temporal or spatial locality, and treat these entries differently from the other entries.

An example of this strategy is Low Inter-Reference Recency Set (LIRS) [Jiang and Zhang, 2002]. Instead of using recency directly like LRU, LIRS uses Inter-Reference recency (IRR), also called reuse distance. IRR refers to the number of other entries accessed between two consecutive references to a given entry. This allows the cache entries to be divided into either hot or cold entries. Hot entries have a low Inter-Reference recency and are likely to be accessed again in future. They should therefore remain in the cache. And cold entries, on the other hand, have a high Inter-Reference recency and are unlikely to be accessed again in future, which means that they can be easily removed without causing many additional cache misses. So we divide the cache capacity into a smaller part for hot entries and a larger part for cold entries. Furthermore, a variable number of non-resident cold entries can be stored. The non-resident cold entries are necessary to track the IRR. LIRS uses one LRU priority queue and one FIFO queue, as shown in Figure 2.4. The LRU-queue stores all hot, all resident cold and non-resident cold entries. The FIFO queue stores only the resident cold entries and is used to find the entry to be evicted. Instead of tracking the IRR directly, the position between entries in the LRU priority queue is used to determine the IRR and to to control when a cold entry converts to a hot entry and vice versa. The advantage of LIRS is that it eliminates the weakness of LRU for weak locality workloads with relatively low overhead. The disadvantage is that we still have to predefine the ratio between the capacity for hot and cold entries.

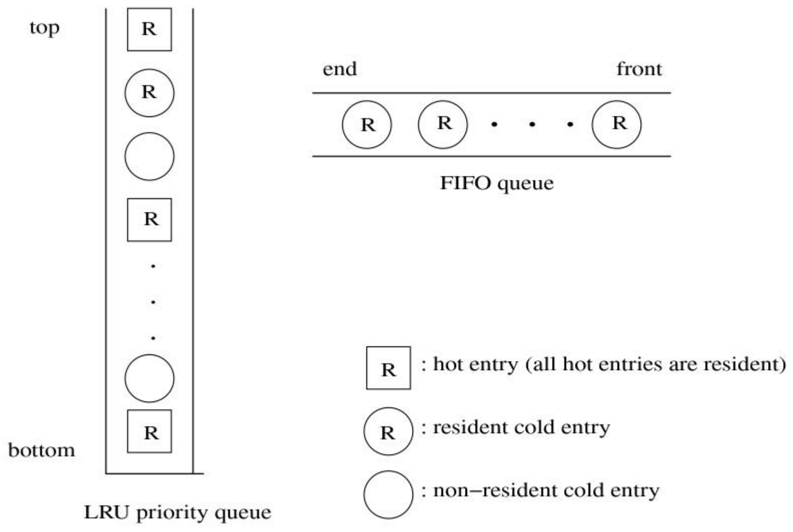

top
bottom
LRU priority queue
end
front
FIFO queue
R : hot entry (all hot entries are resident)
R : resident cold entry
: non−resident cold entry

Figure 2.4.: Shows the LRU stack and FIFO queue used for LIRS, original from [Jiang and Zhang, 2002] labeling altered.

Another example is Adaptive Replacement Cache (ARC) [Megiddo and Modha, 2003]. ARC uses two LRU priority queues, L_1 for entries who are accessed only once in the recent past and L_2 for entries accessed at least twice in the recent past. So L_1 measures recency and L_2 the frequency. For a cache with the capacity for c number of entries, each queue, L_1 and L_2, can also hold c number of entries. Thus, the cache saves $2c$ entries, of which c are non-resident entries. To achieve adaptability, the number of resident entries for each queue is not fixed and changes according to the workload. Each time a cache hit occurs on a non-resident entry in one of the queues, the number of resident entries for that queue is increased, depending on the ratio between the number of non-resident entries of each queue. Also, whenever a cache hit occurs on an L_1 entry, whether resident or non-resident, it is removed from L_1 and added to L_2. So, depending on the workload and thus the non-resident entries being accessed, ARC behaves more like LRU or LFU. The significant advantage of ARC is that it is adoptable and has no parameters like LIRS or 2Q that need to be manually optimized. Also, the space overhead is still low with only 0.25 percent of the cache size [Megiddo and Modha, 2004].

2.5.4. Using Machine Learning

The last strategy has emerged only in recent years and seeks to incorporate machine learning techniques to learn the underlying access pattern to increase the hit rate. This can be done either by online learning, where learning occurs during execution and is adoptable to changing workloads, or by offline learning, where parameters are learned before execution using a predefined data set.

An example of this strategy is fuzzy page replacement algorithm by [Akbari Bengar et al., 2020], which uses fuzzy c-means, an unsupervised machine learning technique, to find clusters in all cache entries based on the frequency, recency and time difference between two consecutive accesses for each entry. Each time an entry needs to be evicted, the entry with the greatest distance to all cluster centers is evicted.

Another example is proposed by [Choi and Park, 2022] which uses Seq2Seq network, a type of recurrent neural network designed for handling sequence data. The history of the last accessed cache entries is used to predict a sequence of future accesses. The predicted sequence is then applied to conventional cache replacement algorithms, like LFU or LRU, to prevent unnecessary evictions.

2.6. Summary

The research question of this thesis concerns cache performance for a copy on write optimized hierarchical storage stack. As an example of such a storage stack, we use Haura, which uses B^{ε}-tree, a write optimized version of B-tree, as the underlying data structure. The management of the data in Haura is centrally performed by the DMU, the Data Management Unit. For this purpose, the DMU uses a cache containing the data that is currently being used.

The main difficulty with caches is how to decide which entry to evict. Because evicting entries that will be needed in the near future leads to further cache misses and thus to lower cache performance. Therefore, cache replacement policies are used to decide which entry should be evicted from the cache. Cache replacement policies can use additional data structures or per-entry metadata to guide their decision. Various cache replacement policies have been developed, which can range from rather simple queue based policies to policies based on complex heuristics or machine learning.

We looked at four early developed policies. The first was the Bélády's algorithm or optimal cache replacement policy, which had only theoretical value and can be used to determine the optimal achievable cache performance after the execution of an application. The next policy was FIFO, which always evicts the entry that has been in the cache the longest. FIFO is easy to implement, but suffers from the Bélády's anomaly and poor performance in many use cases. LRU uses recency to decide which entry should be evicted and shows good performance in cases of strong locality. The disadvantage of LRU is its poor performance under weak locality, such as loop and scan access patterns. The last policy was LFU, which uses frequency information for eviction decisions. LFU has good performance in the case of skewed access distributions and also performs better than LRU on loop and scan access patterns. The disadvantage of LFU is the higher complexity compared to LRU and FIFO.

All four cache replacement strategies we examined have some weaknesses. Therefore, we have considered strategies for improved cache replacement policies. We first considered explicit user level hints, which can be useful in certain cases, but are not portable to different hardware or adoptable to different use cases. Hence, they are not suitable as a general cache replacement policy.

Another strategy we considered was using more historical information. This could either mean tracking more meta data for each cache entry, like LRFU, or the using of ghost entries, which are entries where only the meta data or a reference to the data stays in the cache, like 2Q.

The third strategy we considered was to identify specific access patterns and adopt the behavior of the cache accordingly. One example of this strategy is LIRS, which uses the Inter-Reference Recency to categorize the cache entries in hot and cold entries. LIRS keeps the hot entries in the cache and prefers to evict the cold entries. Another example is ARC, which uses two queues. One for entries that have been accessed once in the recent past, and a second queue for entries that have been accessed at least twice in the recent past. The ratio between stored entries for both queues is not fixed and a heuristic is used to decide which queue should be stored more entries. This allows ARC to be adapted to different use cases.

The last strategy we considered, using machine learning to make eviction decisions, is a relatively new trend that has emerged in recent years. The used ml-techniques range from simple models with low overhead, which are suitable for online learning, to more sophisticated techniques with pre-trained neural networks.

Overall, cache replacement policies are a current research topic, which is becoming even more important due to the widening processor memory gap. New opportunities are emerging through the use of increasingly better ML techniques. Nevertheless, there is still no general best policy that works in all use cases and can adapt to changing access patterns.

Chapter 3.

Related Work

In this chapter, we present related work. In addition to Chapter 2, we present current cache replacement policies that are designed to increase scalability and are intended to be used in multi-threaded applications.

It's Time to Revisit LRU vs FIFO by [Eytan et al., 2020]

This paper the authors proposes to reevaluate the assumption that LRU has better performance than FIFO, as stated in [Van Den Berg and Gandolfi, 1992]. There were two main reasons for this reevaluation. First, caches are getting increasingly larger and managing the cache metadata can not be done entirely in the main memory. As a result, the management will be partially shifted to persistent storage media. In this case, FIFO can be the better choice since only the front and the end of the FIFO queue need to be in main memory and no additional metadata has to be updated. This means that the largest part of the FIFO queue can be swapped out of the main memory. The second reason is that the emergence of new workloads could mean that old assumptions no longer apply. Cache replacement policies in the past were designed for workloads around memory, files and block storage. But new workloads for big data and machine learning could have different characteristics. The authors concluded that, especially for large caches, FIFO may be a better choice than LRU because of the lower overhead. However, for more traditional workloads where the cache fits in the main memory, LRU is still the better choice.

FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion by [Yang et al., 2023]

Although FIFO is a simple to implement cache replacement algorithm with O(1) complexity for insertion and removal and therefore provides good throughput and good scalability. The major drawback was the lower hit rate compared to LRU-based algorithms. The authors of this paper [Yang et al., 2023] show that FIFO can achieve comparable performance to LRU based cache replacement algorithms. To improve FIFO, they used Quick Demotion and Lazy Promotion.

Quick Demotion is based on the observation that often newly inserted entries also leave the cache quickly. To implement this, the cache is divided into a large main FIFO queue that covers 90% of the capacity and a smaller FIFO queue, the QD queue, covers 10% of the cache capacity.

Furthermore, there is a ghost queue with the same capacity as the cache, where the reference to the evicted entries is stored. So newly inserted entries are added to the QD queue. When the QD queue reaches its capacity, the entry at the front is evicted and added either to the ghost queue if it was not referenced during its time in the QD queue, or to the main FIFO queue if it was referenced. Also, if a cache miss occurs and that entry is in the ghost queue, it will be directly added to the main FIFO queue. The idea of Quick Demotion is not new and some cache replacement policies use similar approaches, like ARC or 2Q. However, the difference is that this approach allows for an even quicker eviction of newly inserted entries. Besides, Quick Demotion can be used in conjunction with different cache replacement policies. This way, Quick Demotion is used for newly inserted entries, while the main cache follows a different cache replacement policy, not necessarily a FIFO.

Lazy promotion attempts to keep frequently accessed entries in the cache with minimal overhead. To achieve this, promotion occurs only on eviction, not on cache hits as in LRU and LFU, which reduces the overhead of processing the metadata. An Example for this would be the reinsertion of referenced entries to the FIFO queue, also called second chance policy.

Furthermore, the authors propose modular cache replacement policies where a state of the art cache replacement policy is used as the main cache and other techniques such as Quick Demotion and Lazy Promotion can be added to improve this replacement policy.

FrozenHot Cache: Rethinking Cache Management for Modern Hardware by [Qiu et al., 2023]

In this paper the authors propose a method to increase the scalability for list based cache replacement policies for applications with skewed accesses distribution. For this purpose, the cache is divided into two parts, the frozen cache and the dynamic cache. The frozen cache is intended to contain the hot entries and should improve throughput by eliminating promotion and locking. The dynamic cache, on the other hand, is used for workload adaptation and works like a regular list-based cache, such as LRU or LFU. The FrozenHot cache works in three phases which are repeated periodically. The first phase is the learning phase, which is used to learn the entries that should be placed in the frozen cache. Also, the ratio between frozen cache and dynamic cache capacity is decided during this phase and can adopt to changing workloads. The second phase is the construction phase which is used to build the Frozen Cache. The third phase, which is longer than the other two phases, is the frozen phase. During this phase, the frozen cache remains fixed to allow faster access on cache hits without lock contention. Overall, this is an interesting approach for a scalable cache in multi threaded applications.

TinyLFU: A highly efficient cache admission policy by [Einziger et al., 2017]

The authors of this paper propose a new frequency based cache replacement policy W-TinyLFU, which uses a bloom filter to approximate the frequency information for the cache entries and is specifically designed for skewed access distributions. So, even for large caches that do not fit entirely in the main memory, the bloom filter is small enough to fit and allows the management of the metadata in main memory even for large caches. W-TinyLFU consists of three components. First, the window, a FIFO queue, into which new cache entries are inserted. This queue is relatively small, ranging from 1% of the cache capacity. The second component is the main cache, which makes up the rest of the capacity. The Bloom filter is located between the two. Every time the window queue reaches its limit. The front entry of the window, the window victim, is checked through the bloom filter if this entry should be promoted to the main cache by comparing the window victim with the possible victim entry from the main cache. When the frequency estimation for the window victim is higher than the main cache victim, the window victim will be promoted to the main cache and the main cache victim will be evicted. Otherwise, the window victim will be evicted. The window acts like a Quick Demotion method, so that newly inserted entries can leave the cache quickly. The advantage of W-TinyLFU is that it does not require ghost entries, which reduces the management overhead and space requirements for the cache.

Summary

For this work, we specifically investigated cache replacement policies that can be easily integrated into an existing code base and are also suitable for multi-threaded applications.

The paper by [Eytan et al., 2020] shows that with the emergence of new workloads, such as machine learning, and increasing cache sizes, assumptions such as the independent reference model may not apply. In particular, the authors conclude that FIFO may be a better choice than LRU in cases where the cache does not fit in the main memory.

The work of [Yang et al., 2023] is interesting because not only does it present a cache replacement policy based on simple FIFO queues, which is easy to implement and scalable, but also because the approaches of Quick Demotion and Lazy Promotion can be applied to other cache replacement policies. For future work, Quick Demotion could be added to the already implemented cache replacement policies.

The FrozenHot cache by [Qiu et al., 2023] achieves scalability by periodically "freezing" a portion of the cache. This a new approach that should be investigated further. Also, FrozenHot can be combined with different list-based cache policies, such as LRU or LFU.

The last cache replacement policy we considered was W-TinyLFU by [Einziger et al., 2017], which is a scalable variant of LFU. The better scalability compared to LFU is achieved by compactly handling the metadata in a bloom filter. Thus W-TinyLFU should be considered for large caches with skewed access distribution.

Chapter 4.

Design and Implementation

In this chapter, we describe the cache replacement strategies that we implemented in Haura and also the changes we had to make to Haura to implement them. First, we consider the DML state cycle (Section 4.1), since the behavior of the DMU, and thus the cache, follows it. After that, we are looking at the cache trait, the interface that any cache implementation must follow. Lastly, we examine all the cache replacement strategies that have been implemented. Starting with CLOCK, which was already implemented, followed by the newly implemented cache replacement policies GCLOCK, CLOCK-Pro and ML-CLOCK.

4.1. DML State Cycle

The task of the DML is to manage objects for the tree layer. This includes reading objects from the disk, storing them in the main memory, tracking changes and writing them back. This is done by a single DMU which is shared between all trees of the upper layer. Thus, the DMU is also responsible for the management of the cache.

To accomplish all this, objects must be uniquely identifiable. All unmodified and all on-disk DML objects are identified by an *ObjectPointer*.

```
pub struct ObjectPointer<D> {
    pub(super) decompression_tag: DecompressionTag,
    pub(super) checksum: D,
    pub(super) offset: DiskOffset,
    pub(super) size: Block<u32>,
    pub(super) info: DatasetId,
    pub(super) generation: Generation,
}
```

Listing 4.1: ObjectPointer struct

The struct fields `offset`, `size`, `checksum` and `decompression_tag`, in Listing 4.1, are used to read an object from disk and decompress it. The `info` field can be used to store additional tags for an object. And the last field `generation` is used to track the age of an object.

The DML objects can be called by their unique *ObjRef*. This reference, shown in Listing 4.2, equals to the state of an object at the last access. Thus, four states are possible: unmodified, modified, in write back or incomplete. If an object is unmodified, it can be identified by its

ObjectPointer and when the object is either modified or in write back, objects are given a unique *ModifiedNodeId* for identification. The incomplete state, as seen in Listing 4.2, is an intermediate state achieved only by either serialization during evicting an object to disk or deserialization during fetching an object from disk. Any object in incomplete state reaches either the on-disk state or the unchanged state in the DML cycle. The incomplete state is only used to control internal processes within Haura.

```
pub enum ObjRef<P> {
    Incomplete(P),
    Unmodified(P, PivotKey),
    Modified(ModifiedObjectId, PivotKey),
    InWriteback(ModifiedObjectId, PivotKey),
}
```

Listing 4.2: ObjRef enumeration, parameter P equals ObjectPointer<D>

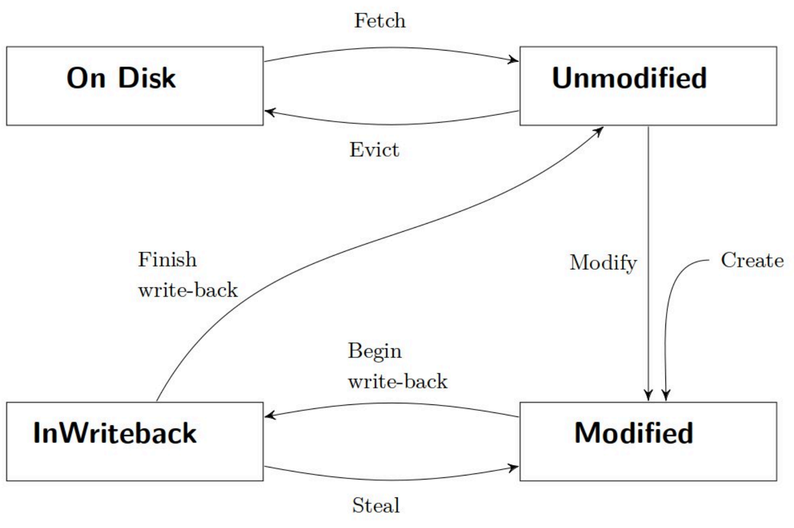


Figure 4.1.: DML state cycle, [Wiedemann, 2018]

The DMU manages the cache and the states of each *ObjRef* as shown in 4.1. The cycle starts with the creation of an object in modified state. When the DMU starts the write back of the object, the state is changed to in write back state. If the object is modified during write back through the steal function, it changes the state back to modified. When the write back is finished, the object state changes to unmodified. An unmodified object can either change the state to modified if the object has been modified or evicted from cache to disk. And lastly, any object on disk can be fetched in cache and will be in unmodified state.

For the cache, all objects are managed with a unique *ObjectKey*, as shown in Listing 4.3. These *ObjectKey* are from the *ObjRef* for each cache entry. Objects in incomplete state are not stored in cache.

```
pub enum ObjectKey<G> {
    Unmodified { offset: DiskOffset, generation: G },
    Modified(ModifiedObjectId),
    InWriteback(ModifiedObjectId),
}
```

Listing 4.3: ObjectKey enumeration

4.2. Cache Trait

The DML state cycle and the fact that the cache is managed by the DMU place additional demands on the cache. We see this in the cache trait, Listing 4.4.

The first difference is that we have an explicit evict function. In most cache implementations, evict is a private function that the cache itself calls. In our case, however, the DMU must call the evict function, not the cache itself. Furthermore, the evict function must take into account that there are cache entries that are pinned i.e. in write back or modified state and therefore cannot be removed from the cache. Also, the DMU can remove specific cache entries, Listing 4.4 line 6 and line 7. The `remove()`-function is used for unmodified entries and `force_remove()` for pinned entries.

```
 1  trait Cache {
 2      fn new();
 3      fn contains_key();
 4      fn get();
 5      fn get_mut();
 6      fn remove();
 7      fn force_remove();
 8      fn change_key();
 9      fn force_change_key();
10      fn evict();
11      fn insert();
12      fn iter();
13      fn size();
14      fn capacity();
15      fn stats();
16      fn verify();
17  }
```

Listing 4.4: Simplified Cache trait without type aliases, generics, function parameters and function return types.

The second difference is that it must be possible to change the key for cache entries. Because an object in cache can change its state. Therefore, we have two functions, Listing 4.4 line 8 and line 9. Again two functions for unmodified and pinned entries.

Another difference is that the cache must be able to handle cache entries of any size. The handling of cache entries of arbitrary size is managed by the DMU. Each fetched entry is always inserted into the cache, and then the eviction function is invoked until the cache capacity is again below the specified cache size. So the cache size is a soft limit that can be temporarily exceeded, not a hard limit, which would mean that the eviction function is called before inserting, so that the cache size is never exceeded.

Furthermore, we have added the `get_mut()`-function, Listing 4.4 line 5, since the `get()`-function does not allow that the state of the cache changes, respectively, atomics are used and thus a read-lock is sufficient.

But for ML-CLOCK, it is necessary that we can change the cache. As we will show in Section 4.3.4, ML-CLOCK uses two linked lists. When a cache hit occurs that also modifies the entry, the modified entry is evicted from one linked list and inserted into the other linked list. This movement cannot be realized with atomic operations. Thus, an exclusive write-lock is necessary for this case. Therefore, we implemented `get_mut()`-function that uses an exclusive write-lock.

4.3. Implemented Cache Replacement Policies

In this section, we examine the implemented cache replacement policies. We have limited ourselves to clock based cache replacement policies since they have a lower overhead and better scalability compared to most list-based cache replacement policies. Since clock-based replacement strategies try to do as little work as possible when cache hits occur and do most of the work during the eviction of an entry, which should be infrequent.

We followed the improvement strategies from the Section 2.5 in selecting the cache replacement policies to implement. The CLOCK policy was already implemented and is an approximation of LRU. The next policy we consider is GCLOCK, a generalization of CLOCK that uses more historical information. CLOCK-Pro is an example of a cache replacement policy that recognizes access patterns. It uses the Inter-Reference recency to distinguish between hot and cold entries and handle these entries differently. The last implemented cache replacement policy, ML-CLOCK, is an example of a policy that uses machine learning. A model is trained based on the removed entries and the cache hits. This model is then used to find the entry for the eviction.

4.3.1. CLOCK

Clock cache was introduced by F.J.Corbató [Corbato et al., 1968] for the Multics operating system and was originally intended to use for the page replacement management for virtual memory. It is an approximation of the LRU (Least Recently Used) policy, but with a low runtime overhead.

Unlike previous cache replacement algorithms, clock uses a circular linked list with a "hand" as a pointer to the current entry. For our implementation in Haura we use a single linked circular list. The hand marks the head of the list and instead of moving the entries through a queue, like in LRU and LFU, the hand is moved. Also, each entry has a reference bit that is initially set to 0 and is set to 1 on a cache hit for that entry.

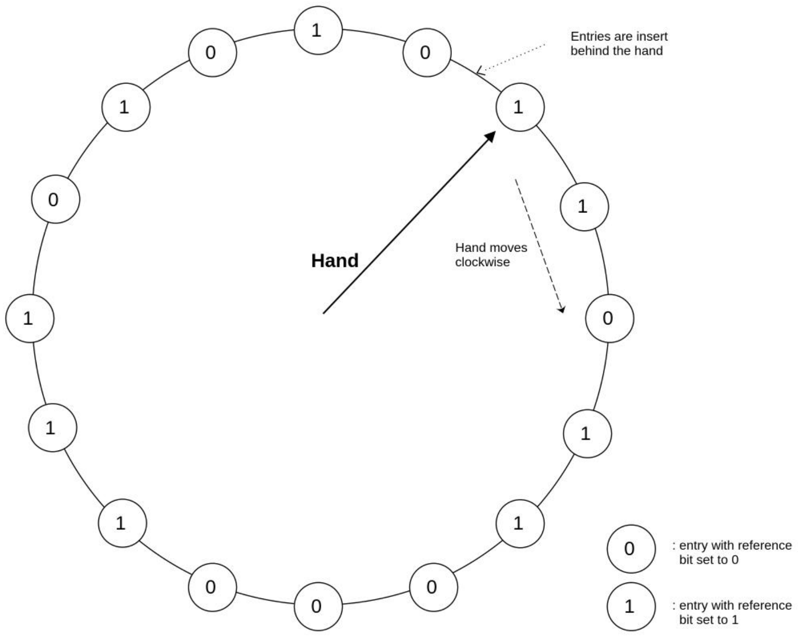

Entries are insert behind the hand
Hand
Hand moves clockwise
1
0
0
1
1
0
1
0
1
1
1
0
1
1
1
0
0
0
0
: entry with reference bit set to 0
1
: entry with reference bit set to 1

Figure 4.2.: Example for circular list in CLOCK.

Listing 4.5 shows the steps during the request of an entry. To find an entry to evict, we look at the entry to which the hand is pointing. If the reference bit for this entry is 0, we have found the entry to be evicted. If the reference bit is 1, we set the reference bit to 0 and move the hand to the next entry. These steps are repeated until an entry is found whose reference bit is set to 0 and which can thus be evicted. New insert entries are placed behind the hand which represents the tail of the list, if we see the hand as the head of the list.

```
Input
    Clk: Clock
    entry: entry on I/O request

if Clk.contains(entry) == true then
    // cache hit
    if entry.referenceBit == false then
        entry.referenceBit = true
    end if
else
    // cache miss
    if Clk.isFull() == true then
        // evict one entry
        while Clk.isFull() == true
            if Clk.hand.referenceBit == true then
                // second chance if reference bit was set
                Clk.hand.referenceBit = false
                Clk.hand = Clk.hand.next
            else
                // found entry for eviction
                evict-candi = Clk.hand
                Clk.hand = Clk.hand.next
                Clk.evict(evict-candi)
            end if
        end while
    end if
    // insert entry to clock
    Clk.insert(entry)
end if
return entry
```

Listing 4.5: CLOCK replacement algorithm in pseudocode.

For our implementation in Haura, we additionally need to handle the case where the entry to be evicted is pinned and therefore cannot be cleared. When this case occurs, we spare the entry, increment the hand, and start the search for the next entry to be evicted from there.

The advantage of CLOCK is that it does not have the problem of lock contention during cache hits, as LRU and LFU do. This is because CLOCK does not have to maintain a priority queue and the update of the reference bit can be done by atomic operations. Therefore, no exclusive lock is required for a cache hit and as a result, CLOCK is more scalable than LRU and LFU in terms of the number of tasks accessing the cache.

Since CLOCK is an LRU approximation, it also has the same disadvantage. The weaker performance on workloads with weak locality, like scanning and looping.

4.3.2. GCLOCK

The GCLOCK replacement policy [Smith, 1978] is a generalization of CLOCK policy. However, the principle is already mentioned for the first time in [Corbato et al., 1968]. Like CLOCK, GCLOCK uses a circular linked list with one hand. But instead of only using one reference bit, a reference counter associated to each cache entry, is used. When a cache hit occurs, the reference counter is incremented up to K times. The parameter K can be chosen arbitrarily. Also, like CLOCK, to find an entry for eviction we look at the entry the hand points to. If the reference counter for this entry is 0, that entry is evicted. Otherwise, we decrement the reference counter and move the pointer to the next entry until we find an entry with a reference counter of 0. The Listing 4.6 shows the steps during the request for an entry.

```
Input
    Clk: Clock
    entry: entry on I/O request
    K: Number of max references stored per entry

if Clk.contains(entry) == true then
    // cache hit
    if entry.referenceCount < K then
        entry.referenceCount += 1
    end if
else
    // cache miss
    if Clk.isFull() == true then
        // evict one entry
        while Clk.isFull() == true
            if Clk.hand.referenceCount > 1 then
                // decrease reference count
                Clk.hand.referenceCount -= 1
                Clk.hand = Clk.hand.next
            else
                // found entry for eviction
                evict-candi = Clk.hand
                Clk.hand = Clk.hand.next
                Clk.evict(evict-candi}
            end if
        end while
    end if
    // insert entry to clock
    Clk.insert(entry)
end if
return entry
```

Listing 4.6: GCLOCK replacement algorithm in pseudocode.

When we choose $K = 0$, GCLOCK works like a FIFO policy. For $K = 1$, GCLOCK works as an approximation to the LRU policy and is the same as the CLOCK policy. Thus, CLOCK is a special case of the GCLOCK policy. If we choose $K > 1$, we get an approximation to the LFU policy, but with the addition of a built-in aging mechanism. This is because the reference counter is decremented when the hand passes an entry. For our implementation, we use $K = 2$ because [Corbato et al., 1968] has shown that this is a good trade off between performance and additional overhead.

4.3.3. CLOCK-Pro

The CLOCK-Pro cache replacement policy [Jiang et al., 2005] combines the efficiency from CLOCK policy with the performance improvements from LIRS [Jiang and Zhang, 2002]. The basic idea of CLOCK-Pro is the same as in LIRS, which is to categorize cache entries into hot and cold entries based on Inter-Reference recency. The hot entries always stays in cache and the cold entries allow a quick eviction of rarely accessed entries. Also, like LIRS, CLOCK-Pro uses non-resident cold entries to track history information. However, CLOCK-Pro, unlike LIRS, is adoptable because it does not require a predefined parameter to control the number of hot and cold entries.

To achieve better efficiency than LIRS, CLOCK-Pro combines the LRU priority queue and a FIFO queue of LIRS in a circularly linked list. And instead of using only one hand, as in CLOCK, three hands are used to provide the functions of LIRS, as shown in Figure 4.3. For our implementation, we use a circular doubly linked list instead of a circular singly linked list, like in CLOCK. Because in some cases we need to move a hand one entry backwards (move to predecessor) to correctly implement the motion of all hands. Since the hands should not point to the same entry.

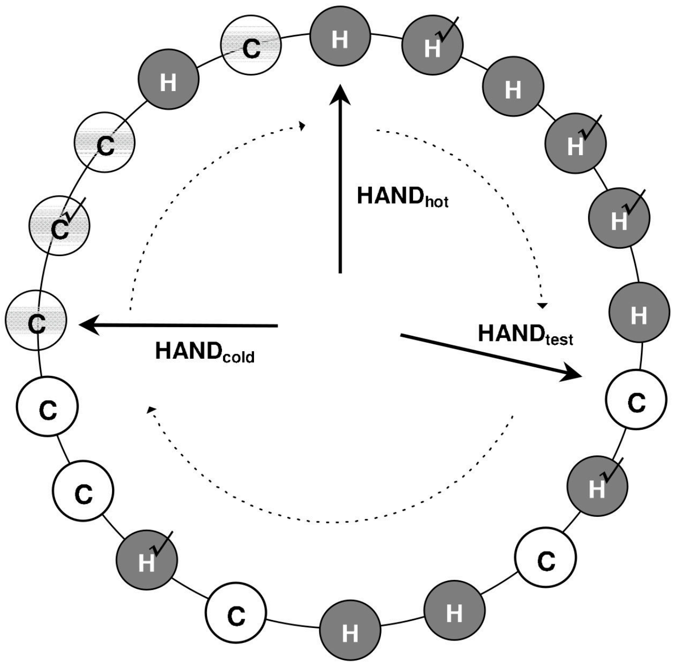


Figure 4.3.: Example of the circular list for CLOCK-Pro. Hot entries marked with "H", cold entries with "C" and non-resident cold entries are shadowed. The "$\surd$" marks referenced entries.

For each entry we need to store additional metadata. This includes whether it is a hot or cold entry, is it resident or non-resident in the cache, has it been referenced in the near past, and is the entry in the test phase. Each hot entry is always resident and not in the testing phase. Cold entries, on the other hand, may be resident or non-resident and may or may not be in the testing phase. Also, all newly inserted entries are resident cold entries in the test phase. Both can be referenced or not. The test phase is used to give cold entries the chance to turn into hot entries. As we will see in a moment.

The $\textbf{Hand}_{hot}$ marks the tail of the list and therefore points to the last hot entry. Also, $\textbf{Hand}_{hot}$ turns unreferenced hot entries into cold entries and removes the reference for referenced hot entries when it passes over them.

The $\textbf{Hand}_{cold}$ is used to find the entry for eviction. In this respect $\textbf{Hand}_{cold}$ operates like the hand in CLOCK. This means that $\textbf{Hand}_{cold}$ will traverse all entries until it finds a resident cold entry that is not referenced and thus can be evicted. If the entry to be evicted is in the test phase, it remains in the cache as a non-resident entry, if it is not in the test phase, it is completely evicted from the cache. So if there are only hot entries in the cache, we must first convert hot entries to cold entries in order to evict an entry. Furthermore, $\textbf{Hand}_{cold}$ turn referenced cold entries in the test phase to hot entries. This is because the reference during the test phase means that the Inter-Reference recency for this entry is low enough to be a categorized as an hot entry.

And the $\textbf{Hand}_{test}$ marks the last entry in the test phase and ends the test phase for all resident cold entries when it moves over them. It also evicts passed non-resident cold entries from the cache. The reason for this is that non-resident cold entries that have not already been converted to hot entries have a high Inter-Reference recency and therefore should not become hot entries.

The Listing 4.7 shows the steps during the request for an entry. What we see is that all hands work together to find an entry for eviction, Listing 4.7 line 12 to line 65.

```
Input
    Clk: Clock
    entry: entry on I/O request

if Clk.contains(entry) == true then
    // cache hit
    if entry.referenceBit == false then
        entry.referenceBit = true
    end if
else
    // cache miss
    if Clk.isFull() == true then
        while Clk.countHot + Clk.countCold >= Clk.capacity
            // run cold-hand to find evict candidate
            evict-candi = Clk.coldHand
            if evict-candi.coldBit == true then
                if evict-candi.referenceBit == true then
                    // turn evict-candi to hot entry
                    evict-candi.hotBit == true
                    evict-candi.coldBit == false
                    Clk.countHot += 1
                    Clk.countCold -= 1
                else
                    // turn resident Cold entry to non-resident
                       ↪ Cold entry
                    // only retain key and metadata
                    Clk.turnNonResident(evict-candi)
                    Clk.countCold -= 1
                    Clk.countTest += 1
                end if
            end if
            // move test hand if necessary
            while Clk.countTest > Clk.testCapacity
                // test hand turns resident cold entry to
                   ↪ non-resident cold entry
                // if still in test-period
                entry = Clk.testHand
                if entry.testBit == true then
                    Clk.testHand = Clk.testHand.next
                    Clk.removeNonResidentEntry(entry)
                    Clk.countTest -= 1
                    if Clk.coldCapacity > 1
                        Clk.coldCapacity -= 1
                    end if
                end if
                Clk.testHand = Clk.testHand.next
            end while
```

```
                // move cold hand
                Clk.coldHand = Clk.coldHand.next
                // move hot hand if necessary
                while Clk.countHot > Clk.capacity - Clk.coldCapacity
                    // hot hand removes reference bit on hot entry
                    // or turn not referenced hot entry to cold entry
                    entry = Clk.hotHand
                    if entry.hotBit == true then
                        if entry.referenceBit == true then
                            entry.referenceBit == false
                        else
                            entry.hotBit = false
                            entry.coldBit = true
                            Clk.countHot -= 1
                            Clk.countCold += 1
                        end if
                    end if
                    Clk.hotHand = Clk.hotHand.next
                end while
            end while
        Clk.insert(entry)
end if
return entry
```

Listing 4.7: CLOCK-Pro replacement algorithm in pseudocode.

For a cache with capacity m applies $m = m_c + m_h$. Here m_c is the capacity for resident cold entries and m_h is the capacity for hot entries. Furthermore, at most m non-resident cold entries can be stored. So that the cache's actual size can be at most $2m$ entries. To achieve adaptability, the number of hot and cold entries is not fixed. The capacity for resident cold entries, m_c is increased by one, if a cold entry is accessed during its test phase and m_c is decreased by one, if a cold entry leaves its test phase without being re-accessed. As we see in Listing 4.7 the actual hand movements are triggered by the count of hot and cold entries and the capacity for hot and cold entries. This changes the behavior of CLOCK-Pro depending on the underlying workload.

When m_c is large, CLOCK-Pro behaves similarly to LRU. But if m_c is small it behaves like LIRS. The small m_C leads to a quicker eviction of newly inserted entries and the entries with low Inter-Reference recency, hot entries, have a higher chance to stay in cache. Thus, in theory, CLOCK-Pro has the advantage of LRU for workloads with strong locality and the advantages of LIRS for workloads with weak locality.

In our implementation in Haura, we have to modify the CLOCK-Pro, due to the fact that the DMU manages the cache. So we can not turn non-resident cold entries to hot entries, which are always resident. Because this would imply that the cache tells the DMU what to do and not the other way around.

Furthermore, the treatment of a pinned entry during eviction, is more complicated than for CLOCK and GCLOCK. We have already seen that for eviction all pointers work together. If the entry intended for eviction remains in the cache, it could happen that a hand gets stuck on this entry, since the entry is pinned it cannot be removed. This can lead to an endless loop. To prevent this, this entry is removed and temporally saved. So that the hands can work as intended. If it then occurs that the entry is pinned, the entry is again inserted into the cache.

4.3.4. ML-CLOCK

The last cache replacement policy implemented is ML-CLOCK by [Cho and Kang, 2021] and follows the recent trend of incorporating machine learning techniques. The idea for ML-CLOCK is to use recency and frequency information for eviction. In order to achieve adaptability, we cannot use a fixed weighting parameter as we have seen with LRFU [Lee et al., 2001]. If the weighting parameter is fixed, we cannot adapt to the access pattern. So, instead of using a fixed parameter to decide what is more important, recency or frequency, ML-CLOCK learns the weighting parameter from previous decisions and is therefore adaptive.

In addition, ML-CLOCK is designed to minimize writes to the underlying storage devices. The motivation for this is that write operations take more time than read operations. Also, some memory types, like flash memory, can only write in blocks. Thus, we can minimize write operations, if we manage to write back adjacent data blocks, instead of writing back data scattered across a storage device.

For each entry we have a reference bit, a reference counter and a time stamp of the last accesses as metadata. Also, the entries are categorized into clean (unmodified) entries and dirty (modified) entries. And in contrast to the other implemented cache replacement policies, ML-CLOCK uses two circular linked lists. One for the clean entries which we refer to as clean-clock and a second for the dirty entries, which we will refer to as dirty-clock which is also sorted by the logical block address for each entry. Both circular lists have one hand, the c-hand for the clean-clock and the d-hand for the dirty-clock. When the d-hand moves to the next entry, it moves to the next entry which has the closest address in the dirty clock. This ensures that sequential blocks are written back, if possible. Furthermore, we use a ghost queue which saves the metadata of evicted entries and is used to train the single-layer perceptron(SLP). At most, the ghost queue can store as many entries as the cache, clean-clock and dirty-clock together.

ML-CLOCK uses a single-layer perceptron(SLP), as a binary classifier, which in our case predicts whether an entry should be evicted or not. One advantage of SLP compared to more advanced machine learning techniques, such as multi-layer perceptrons or long short-term memory networks, is that SLP has low time and space overhead, making it more suitable for online learning. Furthermore, SLP are not prone to overfitting.

To predict if an entry will be evicted we use equation 4.1, where w_d, w_c and w_b represents the weight for reuse-distance, reference count and the bias. The variables x_d and x_c represents the metadata values from the entry reuse-distance and the reference count.

$$f_{predict}(x_d, x_c) = \begin{cases} 0, & x_d \cdot w_d + x_c \cdot w_c + 1 \cdot w_b < 0 \\ 1, & x_d \cdot w_d + x_c \cdot w_c + 1 \cdot w_b \geq 0 \end{cases} \tag{4.1}$$

Also, x_d must be scaled down so that it has the same order of magnitude as x_c, as shown in equation 4.2.

$$x_d = \frac{\textit{current timestamp} - \textit{timestamp of the last access}}{\textit{number of all entries in cache}} \tag{4.2}$$

In addition to learning the weights w_d, w_c and w_b, the equation 4.3 is used.

$$\begin{aligned} w_d &\leftarrow w_d + lr \cdot x_d \cdot (v_{expect} - v_{predict}) \\ w_c &\leftarrow w_c + lr \cdot x_c \cdot (v_{expect} - v_{predict}) \\ w_b &\leftarrow w_b + lr \cdot x_b \cdot (v_{expect} - v_{predict}) \end{aligned} \tag{4.3}$$

The learning rate lr controls how much the weights change during training. The variable $v_{predict}$ is the result of a prediction, equation 4.1, for an entry and v_{expect} is the correct answer for this prediction, as we see in listing 4.5. Both $v_{predict}$ and v_{expect} can either be 1 or 0. Thus, the term $(v_{expect} - v_{predict})$ defines if the weights are updated or not.

Listing 4.5 shows the steps during the request for an entry and also shows when the prediction and the learning take place.

```
Input
    Clk: Clock
    hand: hand is the entry the clock hand is pointing to
    entry: entry on I/O request
    GhostQ: Ghost Queue

if Clk.contains(entry) == true then
    updateMetadata(entry)
    // trigger learning operation
    // second parameter for learning() is $v_{expect}$
    learning(entry, 1)
    return entry
else
    // cache miss
    if Clk.isFull() == true then
        // scan clean pages at the C-hand's location
        C-candi = Clk.findCleanCandidate()
        // scan dirty pages at the D-hand's location
        // in a sequential order (i.e, LBA)
        D-Candi = Clk.findDirtyCandidate()
        victim = prediction(C-candi, D-candi)
        // check if victim is in ghost queue
        if GhostQ.findPage(victim) == true then
            // ready to promote a entry to clock
            // second parameter for learning() is $v_{expect}$
            learning(entry, 1)
            GhostQ.deleteEntry(victim)
        else
```

```
                if GhostQ.isFull() == true then
                    // evict a page on ghost queue with learning
                    // second parameter for learning() is $v_{expect}$
                    learning(GhostQ.tail, 0)
                    GhostQ.deleteEntry(GhostQ.tail)
                end if
                GhostQ.insertEntry(victim)
            end if
            Clk.evict(victim)
        end if
        // insert entry to clock
        Clk.insert(entry)
    end if
    return entry
```

Listing 4.8: ML-CLOCK replacement algorithm in pseudocode, slightly modified from [Jiang et al., 2005]

On a cache hit, we update the metadata for the entry and start a learning operation. We set v_{expect} to 1, because the requested entry is contained in the cache. Then we use the metadata of the entry to make a prediction, equation 4.1, and get $v_{predict}$. The weights are updated or not based on v_{expect} and $v_{predict}$. In addition, a cache hit may cause a clean entry to become a dirty entry. Therefore, the entry is moved from the clean-clock to the dirty-clock. For this case, we added the `get_mut()`-function to the Cache trait.

To find an entry for eviction during a cache miss, we search for eviction candidates in both clocks. Like the hand in CLOCK, the c-hand and d-hand search for an entry with the reference bit set to 0. So there are two eviction candidates, the c-candidate from the clean-clock and the d-candidate from the dirty-clock. The next step is to apply the prediction operation to both candidates and to decide which candidate to evict the preference rules in Table 4.1 will be used. Since the goal is to minimize write operations, in 3 out of 4 possible cases the c-candidate is removed and not the d-candidate.

Predicted Value (c-candidate)	**Predicted Value (d-candidate)**	**Victim Entry**
0	0	C-Candidate
0	1	C-Candidate
1	0	D-Candidate
1	1	C-Candidate

Table 4.1.: Preference rules for victim entry [Cho and Kang, 2021]

When the entry to evict was found, it is checked whether this entry is already in the ghost queue. If this is the case, a learning operation is triggered. Additionally, the entry is evicted from the Ghost queue and added back to the cache. In our implementation, we do not reinsert the entry into the cache for the same reason as in CLOCK-Pro, the DMU manages which entries are inserted into the cache and not the cache itself.

In case the entry for eviction is not already in the ghost queue, it will be inserted into the ghost queue. A learning operation is triggered if the ghost queue exceeds its capacity and thus a ghost entry at the front leaves the ghost queue.

So there are three possibilities to trigger a learning operation during a cache hit, if the entry for eviction is already in the ghost queue or if an entry must leave the ghost queue.

The case that the entry for eviction is pinned can be handled the same way as in CLOCK and GCLOCK, simply leave the pinned entry in the clock and continue the search at the next entry.

4.4. Summary

The basis for the data management in Haura is the DML state cycle. Each object can be in one of the four states: on disk, unmodified, modified or in write back. Also the DML specifies which state changes are possible. In particular, the fact that objects cannot leave or enter the cache when in modified or write back state, the pinned entries, requires significant adjustments to the cache policy.

The implemented cache replacement policies followed the improvement strategies from the Section 2.5. The first implemented policy was CLOCK, which was already implemented. CLOCK is an LRU approximation which uses a circular linked list, instead of a priority queue as LRU does. By using a circularly linked list, CLOCK achieves similar performance to LRU with lower overhead.

The next policy was GCLOCK, which is a generalization of CLOCK, and uses a reference counter instead of only a reference bit. Thus, GCLOCK can use more historical information for the eviction decision.

The CLOCK-Pro replacement policy uses the Inter-Reference recency, a measure for the reuse distance, to separate entries into hot (low reuse distance) and cold (high reuse distance) entries. The hot entries have a high chance of being reused again in the near future and should therefore stay in the cache. The cold entries which have a low chance of being reused again in the near future should be evicted first.

The last implemented policy was ML-CLOCK, which uses a single-layer perceptron to learn from the evicted entries and the cache hits whether an entry should be evicted or not. In contrast to more advanced machine learning algorithms, a single-layer perceptron is easy to implement, because no additional libraries are required. Also, the run time overhead for a single-layer perceptron is low compared to other machine learning algorithms. Furthermore, unlike the other cache replacement policies, ML-CLOCK attempts not only to minimize cache misses, but also to minimize writes to the underlying storage medium. For this reason, ML-CLOCK uses two circular linked lists, one for the clean (not modified) and one for the dirty (modified) entries.

Chapter 5.

Evaluation

In this chapter, we evaluate the implemented cache replacement policies which we described in Chapter 4. First, we will describe the used hardware and software for performance evaluation. After that, we explain how the benchmarks for performance evaluation are constructed and why we chose them. And lastly, we look at the results of the evaluation which we divided into single threaded and multi threaded workloads.

5.1. Setup

The HPC cluster of the Faculty of computer science of the Otto-von-Guericke-University was used for the evaluation. The Table 5.1 shows the hardware of the used node. As secondary storage, we used the home folder, which is shared between all nodes and uses CephFS.

CPU	AMD Epyc 7443
cores per CPU	24
base clock CPU	2.85
RAM	128 GB DDR4
Secondary storage	shared CephFS across all nodes

Table 5.1.: Hardware used for evaluation.

5.2. Methodology

The performance of a cache strongly depends on the I/O access patterns. Cache replacement policy should exploit this fact by choosing which entries will be evicted and which will remain in the cache to avoid unnecessary cache misses. So in order to test our implemented cache policies, we measure the cache hit rate for each cache replacement policy for different random I/O access patterns.

Fio is used to create synthetic random workloads for a fixed size of I/O. The following three workloads for different I/O sizes were selected. First, random 100% read I/O operations, second random 90% read and 10% write I/O operations and third random 50% read and 50% write operations. Furthermore, we benchmark for a single thread and multiple threads performing I/O operations.

We have also considered workloads with more than 50% write operations. However, such workloads that consist of more writes than reads are rather rare. Additionally, the many write operations lead to a strong memory fragmentation, which either crashes the benchmark or forces us to perform many synchronizations between fio and haura, resulting in a significant increase of the execution time. This caused that we omitted benchmarks with more than 50% write operations.

The selected patterns for random I/O accesses are the following distributions. First, the zipf-distribution (with $\theta = 1.1$) which is a highly skewed long-tailed distribution. This means that only a few objects are accessed frequently and most objects are accessed rarely. It is therefore a cache-friendly workload. Such an access distribution is common, for example, with web caches and databases.

Next, we have selected the normal distribution. The reason for this is that, compared to the Zipf distribution, the I/O accesses are more evenly distributed and there is a pronounced peak, but it is not as dominant as in the Zipf distribution. Therefore, more cache misses may occur and the decision to evict is more relevant.

The third distribution is a zoned-distribution which is defined by how many accesses should fall within a range of the file. For our case, we defined that 60% of the accesses should be in the first 10%, 30% of he accesses should be in the next 10%, 8% of the accesses should be in the next 10% and 2% of the accesses should be in the last 10%. As with the normal distribution, the accesses in the zoned distribution is more dispersed than the zipf distribution.

The last distribution is the uniform distribution. With uniform distribution, the accesses are evenly distributed. This case is therefore the worst case for cache replacement policies, because there is no reason to prefer to keep certain entries in the cache, since all entries are equally likely. Hence, we use this case to evaluate performance under cache-unfriendly workloads.

Listing 5.1 shows an example of a fio jobfile. The random distribution (lines 1-3), thread count (line 7) and size of I/O (lines 12-13) vary depending on the benchmark. The file size is twice the actual size for I/O because, with the use of copy on write, we have fragmentation and if we choose the file size to low the benchmark run can crash. The same random seed is used for each benchmark run (lines 5-6) and the block size for the I/O operations is 1MiB (line 9). The fsync parameter (line 14) defines that after 131072 writes fio will synchronize the data, so that all data that is currently copied for write will actually be written back. This is required to reduce fragmentation.

In A.1, we show the configuration file for Haure. The cache size has been set at 1GiB, which is relatively small to better represent the performance of the different cache replacement policies. If we set the cache capacity higher, the hit rate would be 100% in most cases, or we would have to use very large I/O sizes to actually be able to detect differences.

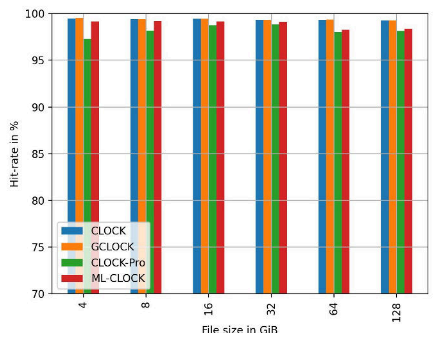
```
[test01]
rw=randrw
rwmixread=90
random_distribution=zipf
randrepeat=1
randseed=937162211
numjobs=4
thread
bs=1m
direct=1
ioengine=external:/.../fio-engine-haura.o
size=16g
filesize=32g
fsync=131072
```

Listing 5.1: Example of a fio jobfile used for evaluation.

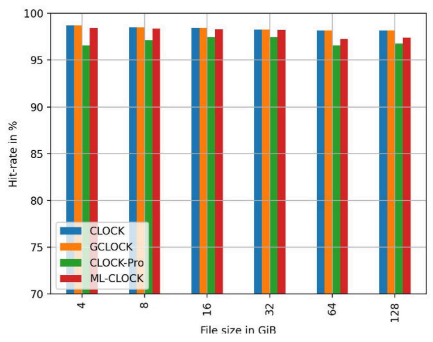
5.3. Single threaded

5.3.1. Random read only Benchmark

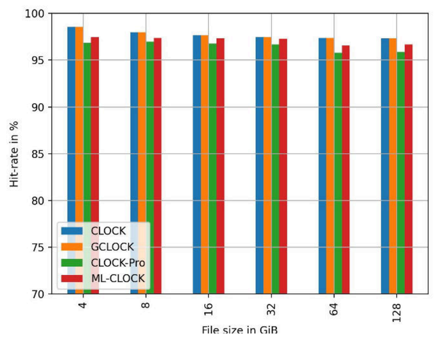
The Figure 5.1 shows the benchmark for random read only workload for the different random distributions. For this case, the hit rate for CLOCK and GCLOCK is nearly identical and both perform the best for all random distributions and file sizes. CLOCK-Pro on the other hand, has the lowest hit rate in all cases. One possible reason for the weaker performance of CLOCK-Pro could be that we had to modify CLOCK-Pro for Haura. As described in 4.3.3, non-resident cold entries are not turned into resident hot entries. Since the hot entries in particular are supposed to prevent additional cache misses, this could have a negative effect on the performance in this benchmark. We also notice that the hit rate of ML-CLOCK decreases slightly with increasing file size. This could mean that the accuracy of SLP in ML-CLOCK may decrease for certain workloads.



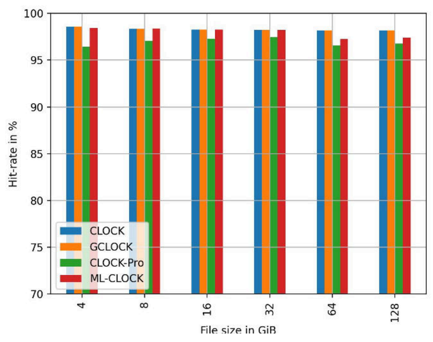
(a) Zipf-distribution



(b) Normal-distribution



(c) Zoned-distribution



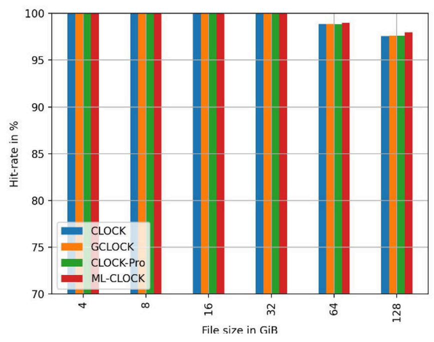
(d) Uniform-distribution

Figure 5.1.: These figures show the hit-rate file-size diagram for each implemented cache replacement policy for random read-only workload with different random distributions.

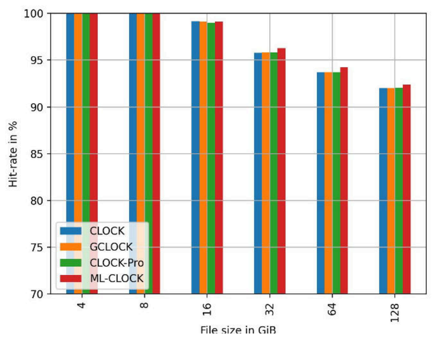
5.3.2. Random 90% read and 10% write Benchmark

The next benchmark is for random 90% read and 10% write operations and is shown in Figure 5.2. In contrast to the read-only case, we have a significantly different result. Partly, this is because with increased write operations more entries are copied for the write and thus are in the modified state and pinned which leads to the decreasing hit rate for larger file sizes.

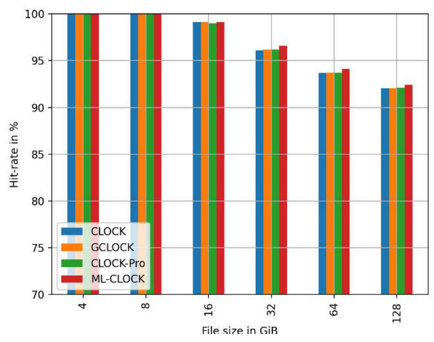
Also, we configured fio to perform synchronization operations after 131072 I/O operations and during this synchronization, all cache entries are deleted. This is necessary because we have already seen in the DML cycle that all objects that are in the status modified or write back are in the cache. So the changes are applied to the data in the cache, not to the actual data on disk. To update all data on disk, each cache entry is dropped and if the cache entry was in modified or write back state, the changes are applied to the original data on disk. To ensure data consistency. Therefore, the cache must be refilled after each synchronization and each newly inserted entry does not count as a cache miss. This leads to a hit rate of 100% for small file sizes. With increasing file size, ML-CLOCK outperforms the other cache replacement policies for all random distributions. It can be assumed that ML-CLOCK has no problems with adapting to the underlying access distribution for this mixed workload even for different random distributions. The performances of CLOCK, GCLOCK and CLOCK-Pro are almost equal across all distributions and file sizes.



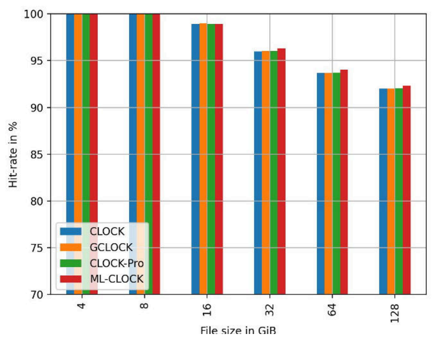
(a) Zipf-distribution



(b) Normal-distribution



(c) Zoned-distribution

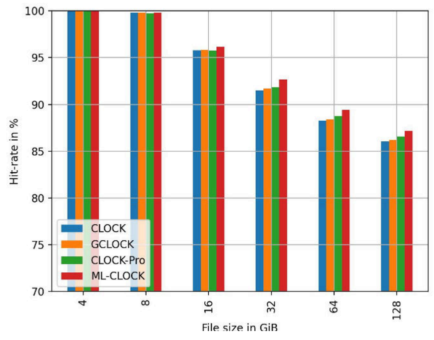


(d) Uniform-distribution

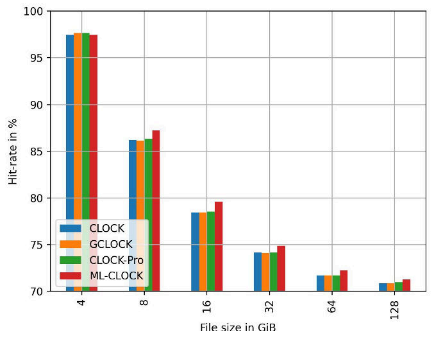
Figure 5.2.: These figures show the hit-rate file-size diagram for each implemented cache replacement policy for random 90% read 10% write workload with different random distributions.

5.3.3. Random 50% read and 50% write Benchmark

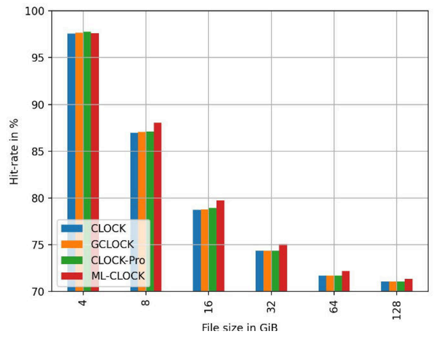
The last single threaded benchmark is for random 50% read and 50% write operations, as shown in Figure 5.3. The first thing to notice is that for all cache replacement policies the hit rates are lower than in the previous benchmark. This is a consequence of synchronization, which occurs more frequently with an increased proportion of write operations, since the cache is flushed with each synchronization. The cache must therefore be rebuilt, which leads to more cache misses. Apart from the generally reduced hit rate, the results are similar to the previous benchmark. ML-CLOCK outperforms the other cache replacement strategies and CLOCK, GCLOCK and CLOCK-Pro again show almost equal performance.



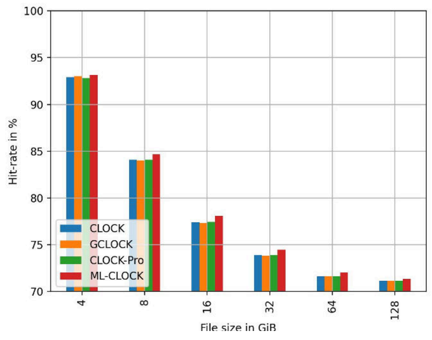
(a) Zipf-distribution



(b) Normal-distribution



(c) Zoned-distribution



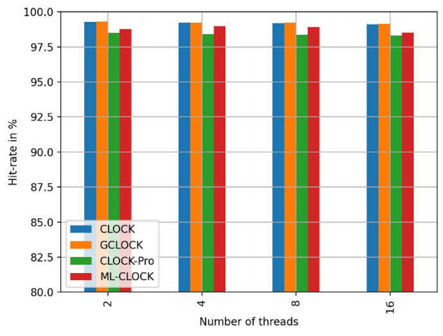
(d) Uniform-distribution

Figure 5.3.: These figures show the hit-rate file-size diagram for each implemented cache replacement policy for random 50% read 50% write workload with different random distributions.

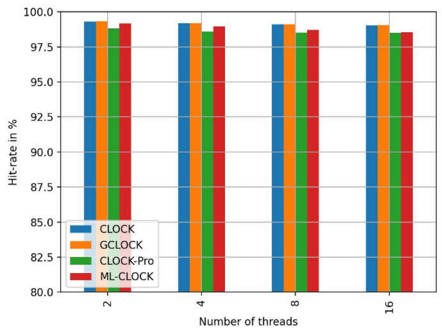
5.4. Multi threaded

5.4.1. Random read only Benchmark

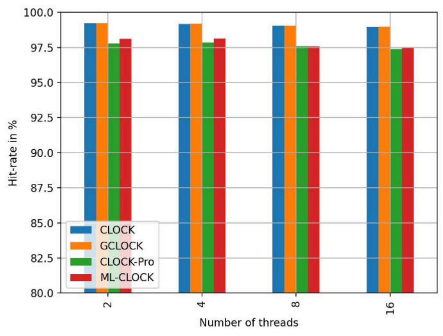
Figure 5.4 to Figure 5.7 show the results for the read only benchmark for the different random distributions at various thread counts. The results have some commonalities with the single-threaded read-only benchmark. As in the single threaded case, CLOCK and GCLOCK have a nearly equal hit rate and outperform the other two cache policies. Also, CLOCK-Pro has the worst performance across all distributions, file sizes and thread counts. This can likely be explained by the same reason as for the single threaded benchmark. The change we had to make to CLOCK-Pro so that we could implement it in Haura.



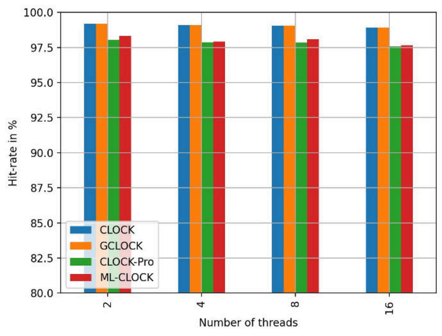
(a) 16 GiB file size



(b) 32 GiB file size

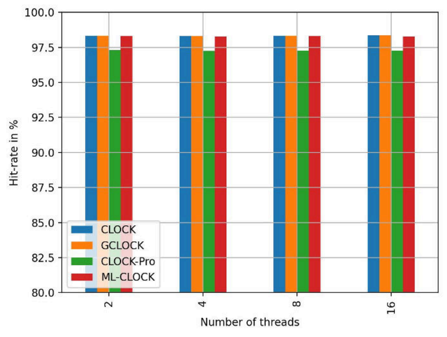


(c) 64 GiB file size

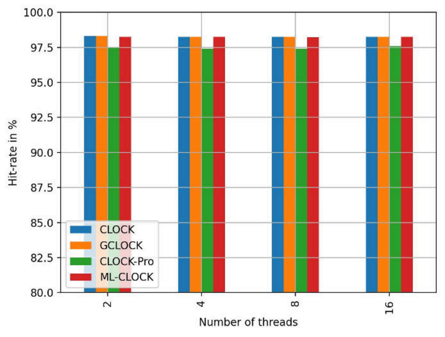


(d) 128 GiB file size

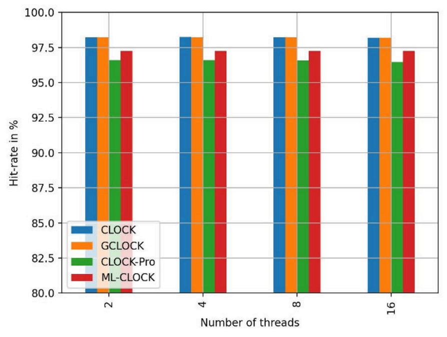
Figure 5.4.: Zipf-distribution, read only workload



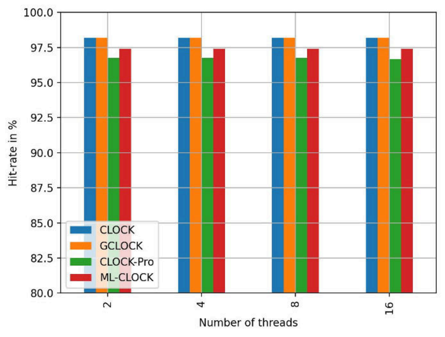
(a) 16 GiB file size



(b) 32 GiB file size

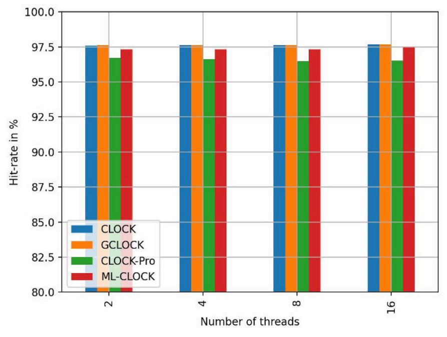


(c) 64 GiB file size

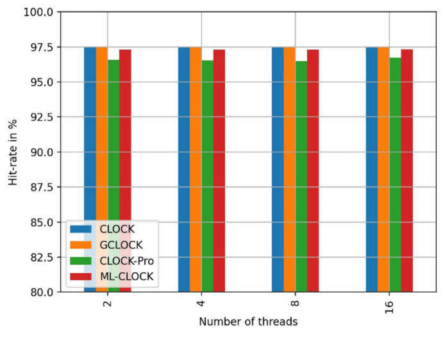


(d) 128 GiB file size

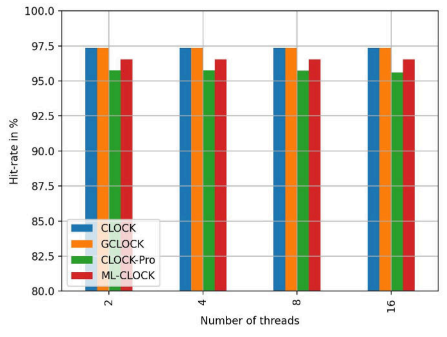
Figure 5.5.: Normal-distribution, read only workload



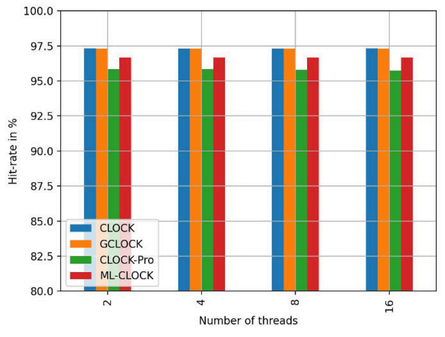
(a) 16 GiB file size



(b) 32 GiB file size

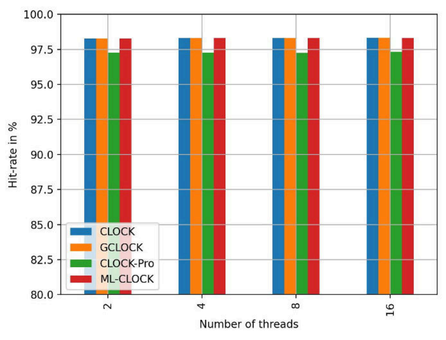


(c) 64 GiB file size

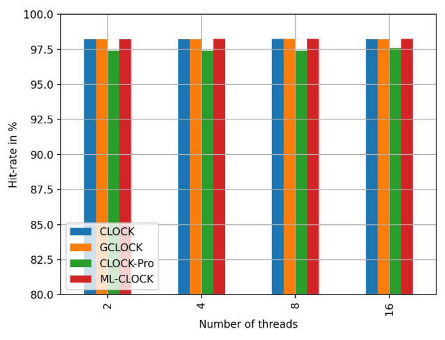


(d) 128 GiB file size

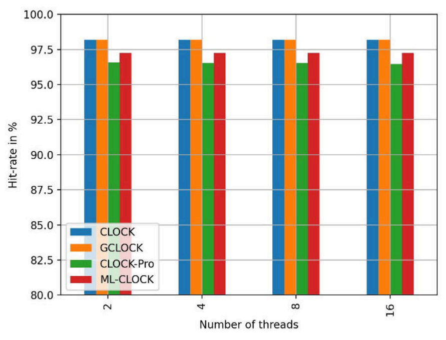
Figure 5.6.: Zoned-distribution, read only workload



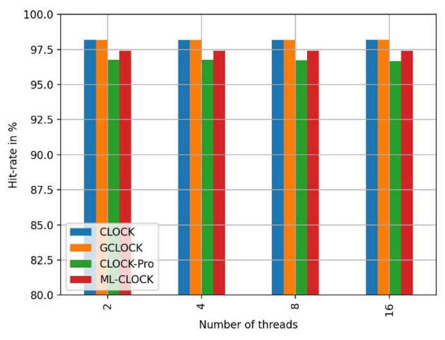
(a) 16 GiB file size



(b) 32 GiB file size



(c) 64 GiB file size



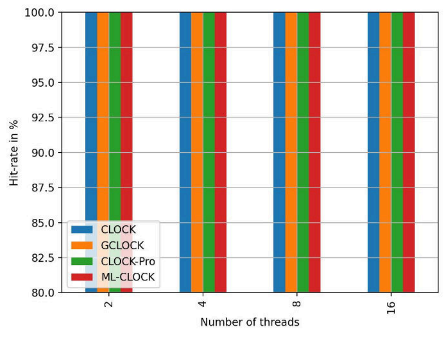
(d) 128 GiB file size

Figure 5.7.: Uniform-distribution, read only workload

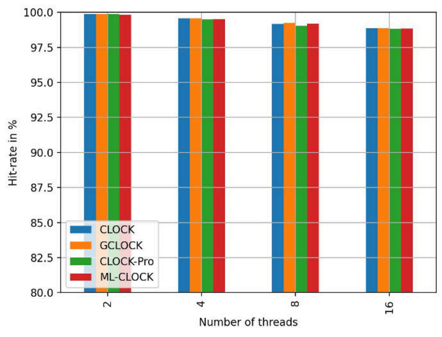
5.4.2. Random 90% read and 10% write Benchmark

For the random 90% read and 10% write Benchmark the results, Figure 5.8 to Figure 5.11, are not as clear as in the read only benchmark. There is no policy that clearly outperforms the other policies.

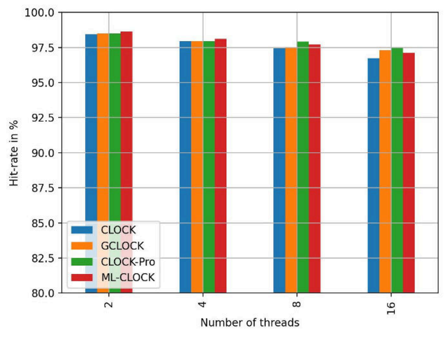
ML-CLOCK has a good performance for low thread counts and for zipf- and normal-distribution. But for 16 threads and uniform distribution, ML-CLOCK shows significantly worse performance than the other policies. CLOCK-Pro, which was outperformed in the single-thread benchmarks, performed best in some cases, specifically Figure 5.10d for 4 threads. Which suggests that the fact that CLOCK-Pro cannot convert non-resident cold entries into resident hot entries, the change we had to make to our CLOCK-Pro implementation, does not significantly affect the performance here. Also, CLOCK and GCLOCK, which had previously shown similar performance, now differ in normal- and zoned-distribution benchmark.



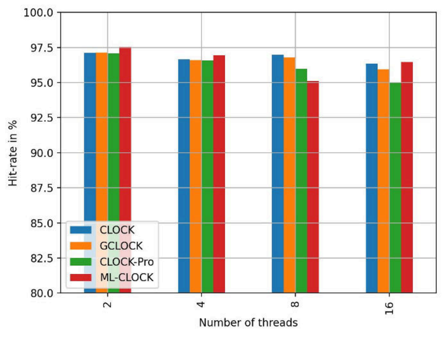
(a) 16 GiB file size



(b) 32 GiB file size

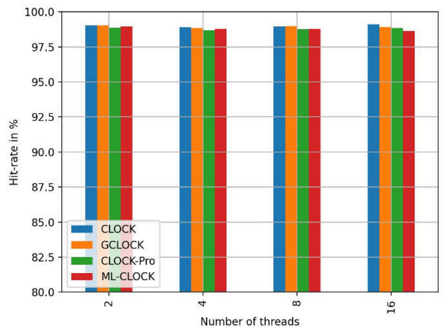


(c) 64 GiB file size

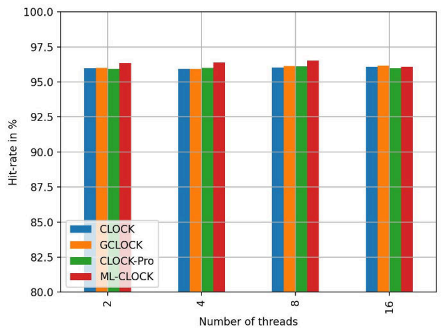


(d) 128 GiB file size

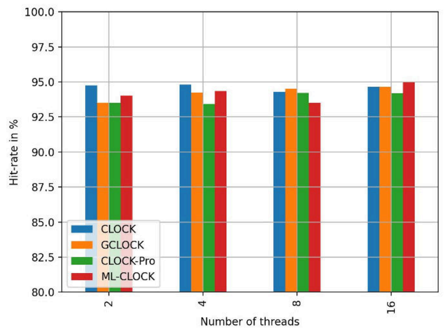
Figure 5.8.: Zipf-distribution, random 90% read 10% write only workload



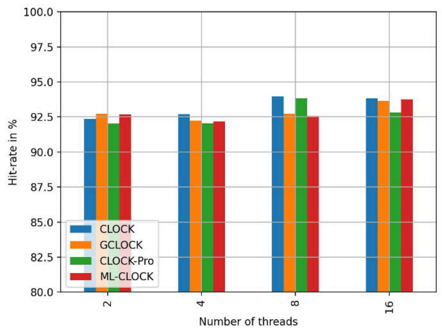
(a) 16 GiB file size



(b) 32 GiB file size

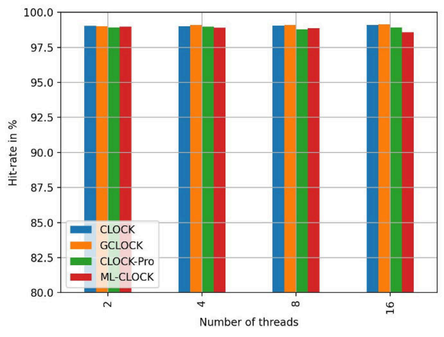


(c) 64 GiB file size

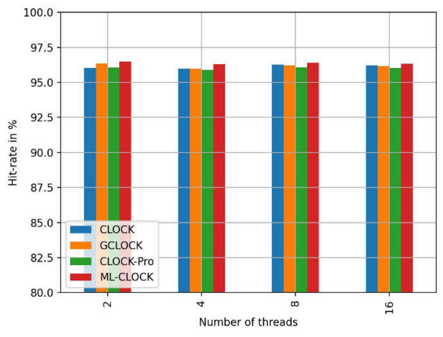


(d) 128 GiB file size

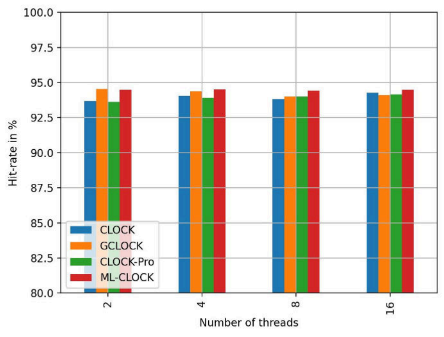
Figure 5.9.: Normal-distribution, random 90% read 10% write only workload



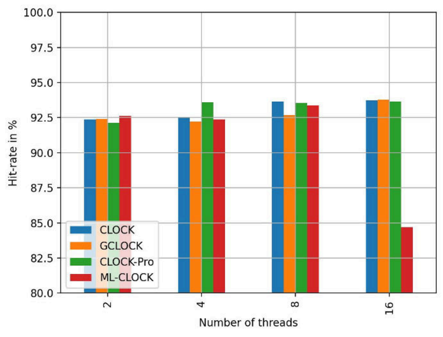
(a) 16 GiB file size



(b) 32 GiB file size

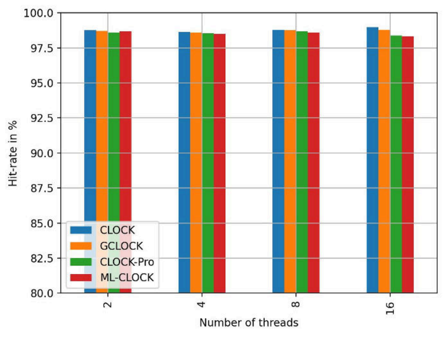


(c) 64 GiB file size

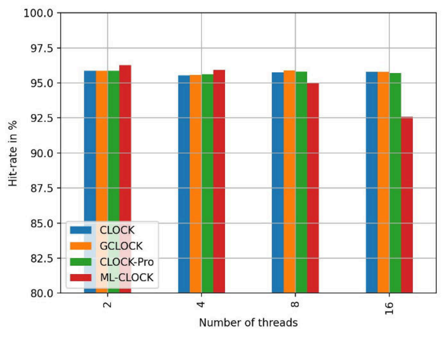


(d) 128 GiB file size

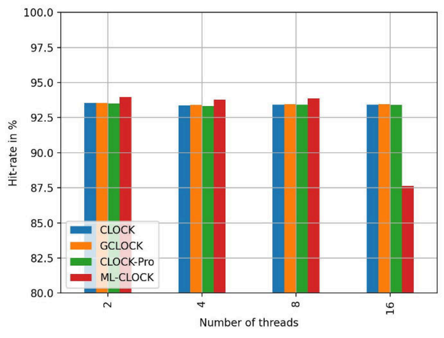
Figure 5.10.: Zoned-distribution, random 90% read 10% write only workload



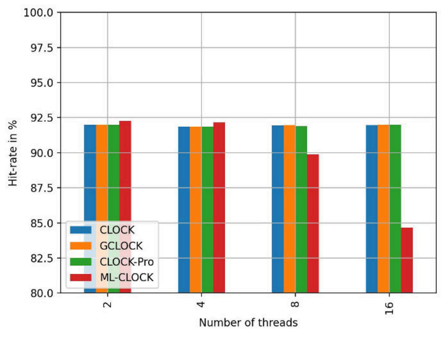
(a) 16 GiB file size



(b) 32 GiB file size



(c) 64 GiB file size



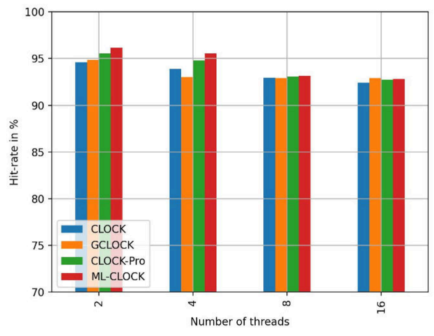
(d) 128 GiB file size

Figure 5.11.: Uniform-distribution, random 90% read 10% write only workload

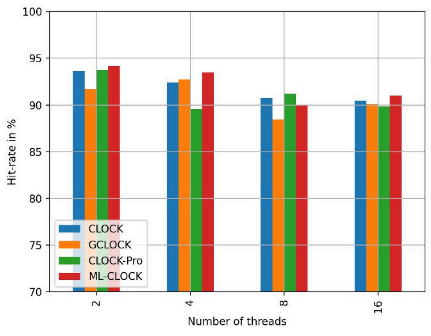
5.4.3. Random 50% read and 50% write Benchmark

The last benchmark is for random 50% read and 50% write operations, Figure 5.12 to Figure 5.15. Here the increased number of write operations again leads to lower hit rates, as in the corresponding single threaded benchmark. CLOCK-Pro again shows better performance for this workload compared to single threaded or the multi threaded read only benchmark. This further supports the theory that in mixed multi threaded workloads our change to CLOCK-Pro is not significant. In particular, CLOCK-Pro performs best at 64 GB and 128 GB file sizes and 8 to 16 threads. ML-CLOCK, on the other hand, has shown a mixed performance. With zipf distribution and low thread count, the performance is good. However, with a higher thread count and file sizes, the performance of ML-CLOCK is usually worse than that of the other cache replacement strategies. A possible explanation for this could be that ML-CLOCK has difficulty learning the underlying access pattern for a higher number of threads and therefore performs worse than the other cache replacement policies in such workloads. Furthermore, the performance of CLOCK and GCLOCK, with the exception of uniform distribution, is more distinct from each other.

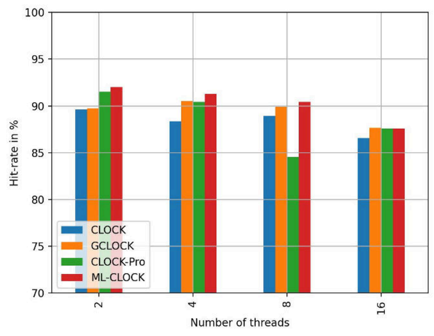
What also stands out for this specific benchmark, with equal ratio of read and write operation Figure 5.15, is that for the uniform random distribution, the hit rate increases with a larger thread count. This is unexpected, since for all other benchmarks the uniform distribution hit rate for all cache replacement polices remains about the same regardless of thread count and file size, with the exception of ML-CLOCK for high thread count. This suggests that the combination of increased thread count and the proportion of write operations improves the hit rate for random uniform distribution.



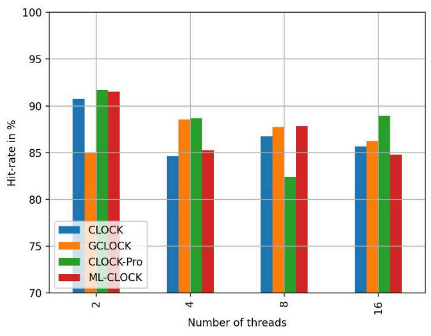
(a) 16 GiB file size



(b) 32 GiB file size

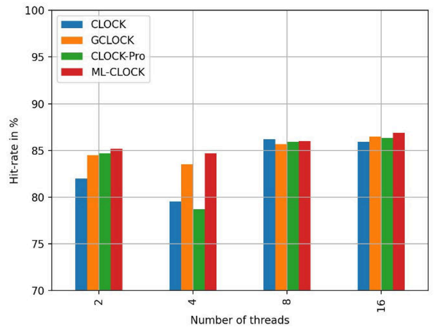


(c) 64 GiB file size

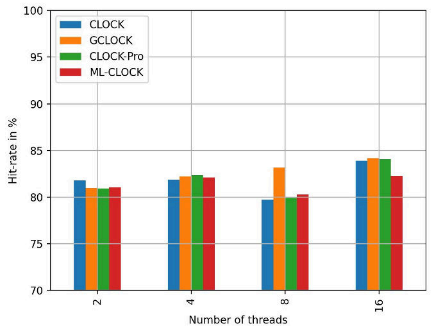


(d) 128 GiB file size

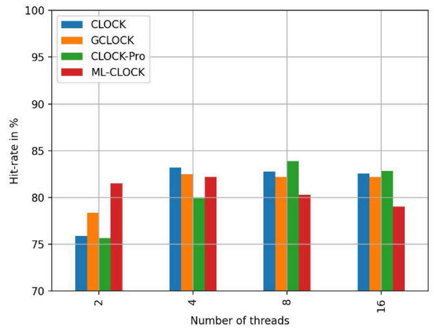
Figure 5.12.: Zipf-distribution, random 50% read 50% write only workload



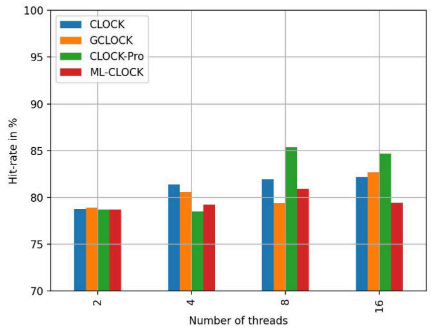
(a) 16 GiB file size



(b) 32 GiB file size

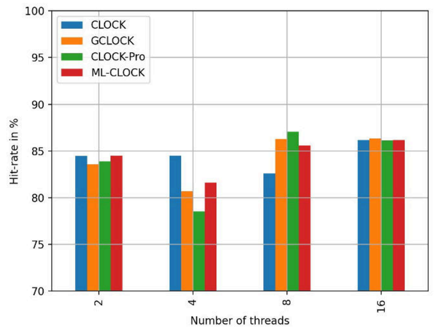


(c) 64 GiB file size

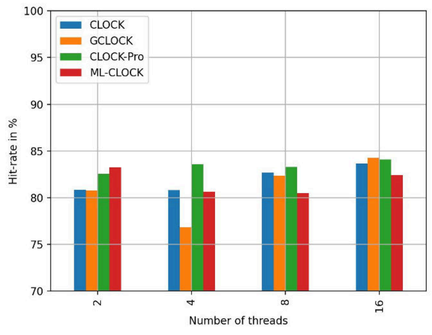


(d) 128 GiB file size

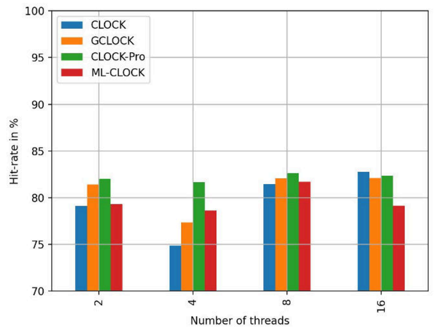
Figure 5.13.: Normal-distribution, random 50% read 50% write only workload



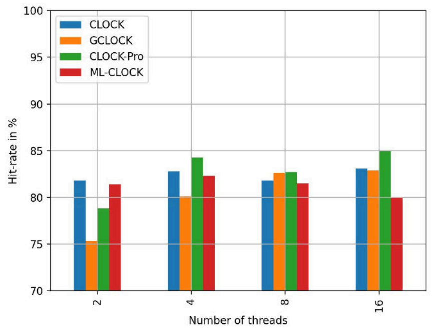
(a) 16 GiB file size



(b) 32 GiB file size

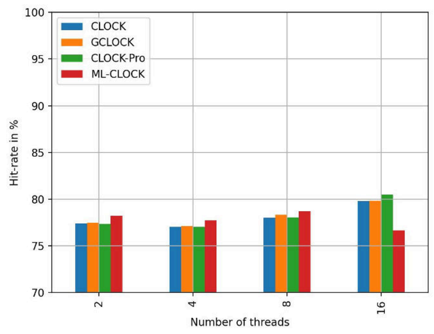


(c) 64 GiB file size

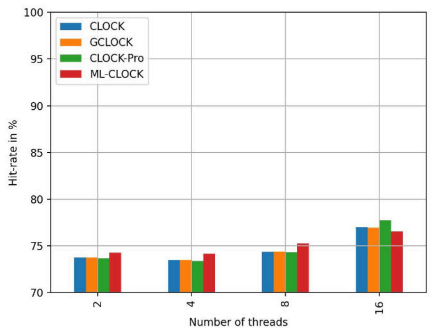


(d) 128 GiB file size

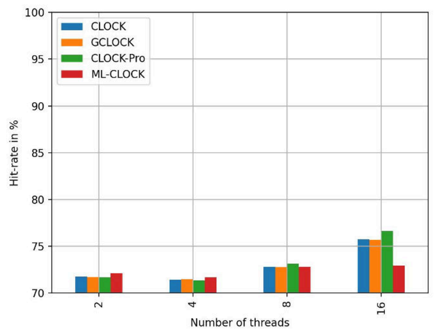
Figure 5.14.: Zoned-distribution, random 50% read 50% write only workload



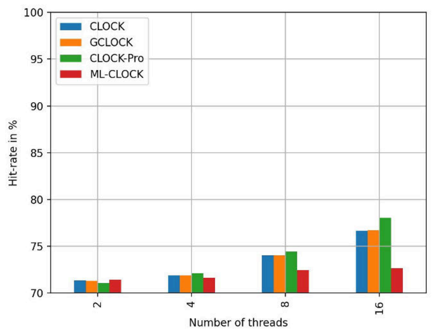
(a) 16 GiB file size



(b) 32 GiB file size



(c) 64 GiB file size



(d) 128 GiB file size

Figure 5.15.: Uniform-distribution, random 50% read 50% write only workload

5.5. Summary

We have considered 3 workloads for the performance evaluation, random reads only and two mixed workloads with different ratios of random reads to random writes for different I/O sizes and measured the hit rate. Four different random distributions were used to simulate different I/O behavior. First, the zipf-distribution as a cache friendly workload that often occurs in web caches and databases. Then the normal- and zoned-distribution, which have a larger dispersion than the zipf-distribution. And last, we used the uniform distribution, which is a cache unfriendly workload since each entry has the same access probability. Furthermore, we evaluated these cases for different thread counts performing I/O.

In the single threaded benchmarks, ML-CLOCK outperforms the other policies for mixed read write workloads. However, CLOCK and GCLOCK performed the best for the read-only workload. CLOCK-Pro, on the contrary, performed worse in the read only workloads but showed comparable performance to CLOCK and GCLOCK in the mixed workloads.

The results of the multi-thread benchmarks are not as clear as those of the single-thread benchmarks. In read only workloads, CLOCK and GCLOCK perform best and CLOCK performs worst, as in the single threaded benchmark. For the mixed workloads, there is no cache replacement policy that outperforms the other. CLOCK-Pro, for example, has good performance at high thread counts and file sizes, but in a few cases has significantly under performance at lower thread counts. Also, the performance of CLOCK and GCLOCK, which was almost the same in the single-thread and multi-thread read-only benchmark, shows significant differences in the mixed multi-thread benchmark.

ML-CLOCK, which showed the best performance for mixed single threaded benchmarks, has significantly lower performance for high thread counts and file sizes. Which could indicate that ML-CLOCK has problems with recognize the I/O access pattern for higher thread counts and file sizes. Another possible cause could be how the ghost queue is managed. Whether a learning operation is triggered depends on if the ghost queue has already reached maximum size and therefore entries have to be removed when a cache entry is evicted from the cache which is then used for learning. The maximum size of the ghost queue depends on how many cache entries are in the cache. So, if a synchronization is performed and the cache is cleared, no entries are removed from the ghost queue for the time being. Only after the cache is refilled and an eviction is performed, thus adding an entry to the ghost queue, the size of the ghost queue is checked. Since the ghost queue is filled by the entries before the synchronization, a large number of learning operations is triggered at once until the size of the ghost queue is again smaller or equal to the size of the cache. This is not a problem for the single threaded benchmark, since there is also only one random distribution for the one thread. But for the multi threaded benchmarks, each thread has its own file to which the random distribution is applied, so the access pattern is more complex. With the triggering of a batch of learning operations at the same time, the SLP may tend to overfitting which is disadvantageous when the access patterns are more complex, as in the case of multi-threaded benchmarks.

The weaker performance of CLOCK-Pro in the read-only benchmarks, both single and multi-threaded, can likely be attributed to our changes to CLOCK-Pro for Haura. The change was that non-resident cold entries cannot be converted to resident hot entries. This change is likely to be more noticeable in read only workloads, as in this case, there is no synchronization which clears the cache. Thus, the cache exists longer and there are more cases where a non-resident cold entry should be converted into a hot entry.

As a result, it can be said that for all implemented cache replacement policies and single threaded use cases ML-CLOCK should be used. CLOCK and GCLOCK have better performance for read only workloads but read only workloads are rather the exception for real applications.

For multi-threaded use cases, it is more difficult to make a recommendation, as the performance of the implemented policies was also mixed and there is no clearly better policy. One possibility could be to use ML-CLOCK for small thread count, about 4 threads, but switch to CLOCK-Pro for higher thread count. Another possibility would be to use CLOCK-Pro for all multi threaded application. However, neither of these options is optimal and may produce suboptimal results for some applications.

Chapter 6.

Conclusion

For this thesis, we implemented and evaluated cache replacement policies for a copy-on-write optimized hierarchical storage stack. We used Haura as the storage stack because Haura is designed as a research storage stack and combines all relevant sub modules from command line interface and data management to benchmark tools in a single code base.

We examined what cache replacement methods are and looked at four early developed and commonly used policies, which were Bélády's algorithm, FIFO, LRU and LFU. Although these cache replacement strategies are still commonly used, they all have some shortcomings and workloads where they under perform. Therefore, we have considered strategies to develop improved cache replacement policies.

We followed these strategies when selecting the cache replacement policies we implemented and evaluated. The CLOCK policy was already implemented and is a variant of LRU. The first policy we implemented, GCLOCK, is a generalization of CLOCK that uses a reference counter instead of a reference bit and follows the strategy of using more historical information. For our second implemented policy, we chose CLOCK-Pro, which follows the strategy of detecting and adapting to access patterns. This is accomplished by dividing the cache entries into hot and cold entries based on their Inter-Reference recency, which are treated differently. The last policy implemented was ML-CLOCK, which uses a single-layer perceptron, a machine learning technique, to learn and adapt to the access pattern.

We also looked at the changes we had to make to Haura to implement the policies. In particular, the circumstance that the DMU plays a central role in the management of the cache meant that the implemented policies had to be modified to ensure the correct functionality of Haura.

To evaluate the different cache policies, we created several benchmarks ranging from random read only workloads to mixed random read write workloads for single and multi-threading. For the random distribution, we used zipf, normal, zoned and uniform distribution to simulate different workload behavior.

In read-only benchmarks, both single and multi-threaded, CLOCK and GCLOCK performed best, and in mixed single-threaded benchmarks, ML-CLOCK outperformed the other strategies. The results are not as clear in the mixed multi-thread benchmarks. CLOCK-Pro, which showed the worst performance in the single-thread benchmarks, shows a significantly better performance in these benchmarks. On the other hand, ML-CLOCK, which had the best mixed single threaded performance, shows significantly worse performance compared to the other strategies, especially in benchmarks with a high thread count and file size.

Furthermore, we have observed that for both single and multi-threaded, the cache hit rate decreases significantly with a larger percentage of write operations. This can be explained by the fact that with an increased number of writes there are also more pinned entries that cannot be removed from the cache.

Based on our measurements, we can say that ML-CLOCK should be used for single threaded use cases. Since it has achieved the best results for all single threaded benchmarks for mixed workloads. In the single threaded read only workloads, CLOCK and GCLOCK performed better, but read only workloads are uncommon for real applications.

It is more difficult to make a recommendation for use cases with multiple threads, since the performance of the implemented strategies was inconsistent and there is no clearly better strategy. An option would be to use ML-CLOCK for a low number of threads and switch to CLOCK-Pro for higher thread counts. A different option would be to use CLOCK-Pro for all multi threaded applications. Nevertheless, both options are not optimal and may lead to suboptimal results for some applications.

6.1. Future Work

This work provides several possibilities for future work. First, the evaluation should be extended to traces of real workloads and not just synthetic benchmarks. Since synthetic benchmarks are only of limited use for predicting performance for application with a non static accesses distribution or very specific access patterns.

Second, we saw that ML-CLOCK was also intended to reduce write operations to the underlying storage medium. For our test system which uses CephFS we could not test if this is actually the case and ML-CLOCK reduces write operations compared to the other implemented policies.

Third, there are various other cache replacement policies that are promising to achieve better performance. In Chapter 2 and Chapter 3 we have presented some further cache replacement policies that can be considered. in particular [Yang et al., 2023] should be implementable with little effort. Since all the implemented cache replacement policies are clock-based and therefore already use some form of Lazy promotion and only Quick Demotion needs to be implemented.

Fourth, the multi threaded workloads in particular cause problems, so this area should be further investigated. One approach could be, instead of using one cache for all threads, to split the cache and give each thread his own cache. This might allow to leverage ML-CLOCK's good performance for single threaded workloads for multi threaded workloads as well.

Fifth, since we have seen that the change we have made to CLOCK-Pro and ML-CLOCK is causing problems and possible performance degradation, the policies and DMU should be modified to correct this.

Bibliography

[Akbari Bengar et al., 2020] Akbari Bengar, D., Ebrahimnejad, A., Motameni, H., and Golsorkhtabaramiri, M. (2020). A page replacement algorithm based on a fuzzy approach to improve cache memory performance. *Soft Computing*, 24(2):955–963. (Cited on page 14)

[Belady, 1966] Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101. (Cited on page 9)

[Belady et al., 1969] Belady, L. A., Nelson, R. A., and Shedler, G. S. (1969). An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353. (Cited on page 9)

[Bender et al., 2015] Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Zhan, Y. (2015). An introduction to b-trees and write-optimization. *login; magazine*, 40(5). (Cited on pages 5 and 6)

[Brodal and Fagerberg, 2003] Brodal, G. S. and Fagerberg, R. (2003). Lower bounds for external memory dictionaries. In *SODA*, volume 3, pages 546–554. (Cited on page 5)

[Cao et al., 1994] Cao, P., Felten, E. W., and Li, K. (1994). Application-controlled file caching policies. In *USENIX Summer*, pages 171–182. (Cited on page 11)

[Cho and Kang, 2021] Cho, M. and Kang, D. (2021). ML-CLOCK: Efficient page cache algorithm based on perceptron-based neural network. *Electronics*, 10(20):2503. (Cited on pages 32 and 34)

[Choi and Park, 2022] Choi, H. and Park, S. (2022). Learning future reference patterns for efficient cache replacement decisions. *IEEE Access*, 10:25922–25934. (Cited on page 14)

[Corbato et al., 1968] Corbato, F. J. et al. (1968). A paging experiment with the multics system. (Cited on pages 24, 27, and 28)

[Danlash, 2022] Danlash (2022). SVG Graphic illustrating Memory hierarchy. `https://commons.wikimedia.org/w/index.php?title=File:ComputerMemoryHierarchy.svg&oldid=686137092`. License: public domain. [Online; accessed 16-October-2023]. (Cited on page 3)

[Denning, 1980] Denning, P. J. (1980). Working sets past and present. *IEEE Transactions on Software engineering*, (1):64–84. (Cited on page 9)

[Efnusheva et al., 2017] Efnusheva, D., Cholakoska, A., and Tentov, A. (2017). A survey of different approaches for overcoming the processor - memory bottleneck. *International Journal of Information Technology and Computer Science*, 9:151. (Cited on pages 1 and 2)

[Einziger et al., 2017] Einziger, G., Friedman, R., and Manes, B. (2017). Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31. (Cited on pages 10, 18, and 19)

[Eytan et al., 2020] Eytan, O., Harnik, D., Ofer, E., Friedman, R., and Kat, R. (2020). It's time to revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. (Cited on pages 17 and 19)

[Ha and Kim, 2022] Ha, M. and Kim, S.-H. (2022). Ccow: Optimizing copy-on-write considering the spatial locality in workloads. *Electronics*, 11(3):461. (Cited on page 5)

[Höppner, 2021] Höppner, T. (2021). Design and implementation of an object store with tiered storage. *Cited on*, page 15. (Cited on page 7)

[Jiang et al., 2005] Jiang, S., Chen, F., and Zhang, X. (2005). CLOCK-Pro: An effective improvement of the CLOCK replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336. (Cited on pages 28 and 34)

[Jiang and Zhang, 2002] Jiang, S. and Zhang, X. (2002). Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42. (Cited on pages 11, 12, 13, and 28)

[Karakostas and Serpanos, 2000] Karakostas, G. and Serpanos, D. (2000). Practical LFU implementation for web caching. *Technical Report TR-622-00*. (Cited on page 10)

[Kuhn, 2017] Kuhn, M. (2017). Julea: A flexible storage framework for hpc. In *High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^ 3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers 32*, pages 712–723. Springer. (Cited on page 7)

[Kumar and Singh, 2016] Kumar, S. and Singh, P. K. (2016). An overview of modern cache memory and performance analysis of replacement policies. In *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, pages 210–214. (Cited on page 4)

[Lee et al., 2001] Lee, D., Choi, J., Kim, J.-H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. (2001). LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361. (Cited on pages 12 and 32)

[Lugar, 2001] Lugar, J. (2001). Hierarchical storage management: leveraging new capabilities. *IT Professional*, 3(2):53–55. (Cited on page 3)

[Mahanti et al., 2000] Mahanti, A., Williamson, C., and Eager, D. (2000). Traffic analysis of a web proxy caching hierarchy. *IEEE Network*, 14(3):16–23. (Cited on page 11)

[Megiddo and Modha, 2003] Megiddo, N. and Modha, D. S. (2003). ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA. USENIX Association. (Cited on page 13)

[Megiddo and Modha, 2004] Megiddo, N. and Modha, D. S. (2004). Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65. (Cited on page 13)

[O'neil et al., 1993] O'neil, E. J., O'neil, P. E., and Weikum, G. (1993). The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306. (Cited on page 11)

[Patterson et al., 1995] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. (1995). Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95. (Cited on page 11)

[Peterson, 2002] Peterson, Z. N. J. (2002). Data placement for copy-on-write using virtual contiguity. (Cited on page 5)

[Qiu et al., 2023] Qiu, Z., Yang, J., Zhang, J., Li, C., Ma, X., Chen, Q., Yang, M., and Xu, Y. (2023). FrozenHot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 557–573. (Cited on pages 18 and 19)

[Rodriguez et al., 2021] Rodriguez, L. V., Yusuf, F., Lyons, S., Paz, E., Rangaswami, R., Liu, J., Zhao, M., and Narasimhan, G. (2021). Learning cache replacement with {CACHEUS}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354. (Cited on page 11)

[Shasha and Johnson, 1994] Shasha, D. and Johnson, T. (1994). 2q: A low overhead high performance buffer management replacement algoritm. In *Proc. 20th Int. Conf. Very Large Databases*, pages 439–450. (Cited on page 12)

[Smith, 1978] Smith, A. J. (1978). Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems (TODS)*, 3(3):223–247. (Cited on page 27)

[Van Den Berg and Gandolfi, 1992] Van Den Berg, J. and Gandolfi, A. (1992). LRU is better than FIFO under the independent reference model. *Journal of applied probability*, 29(1):239–243. (Cited on pages 9, 10, and 17)

[Wiedemann, 2018] Wiedemann, F. (2018). Modern storage stack with key-value store interface and snapshots based on copy-on-write b ε-trees. (Cited on pages 7, 8, and 22)

[Wulf and McKee, 1995] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24. (Cited on page 1)

[Wünsche, 2022] Wünsche, J. (2022). Data migration policies in a copy-on-write tiered storage stack-conception and implementation. (Cited on page 7)

[Wünsche, 2022] Wünsche, J. (2022). Documentation Haura. `https://github.com/julea-io/haura/tree/main/docs`. [Online; accessed 16-October-2023]. (Cited on page 7)

[Yang et al., 2023] Yang, J., Qiu, Z., Zhang, Y., Yue, Y., and Rashmi, K. (2023). FIFO can be better than LRU: the power of lazy promotion and quick demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 70–79. (Cited on pages 17, 19, and 54)

Appendix A.

Appendix

```
{
  "storage": {
    "tiers": [
      {
        "top_level_vdevs": [
          {
            "path": "/home/cgrueneb/.cache/haura/cache.disk",
            "direct": true
          }
        ],
        "preferred_access_type": "Unknown"
      }
    ],
    "queue_depth_factor": 20,
    "thread_pool_size": null,
    "thread_pool_pinned": false
  },
  "alloc_strategy": [
    [
      0
    ],
    [
      1
    ],
    [
      2
    ],
    [
      3
    ]
  ],
  "default_storage_class": 0,
  "compression": "None",
  "cache_size": 1073741824,
  "access_mode": "OpenOrCreate",
  "sync_interval_ms": null,
  "migration_policy": null,
  "metrics": null
}

```

Listing A.1: Haura configuration file used for benchmark runs.

Statement of Authorship

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, October 26, 2023

Signature