UNIVERSITY OF PENNSYLVANIA

UNDERGRADUATE THESIS

# Distributed Caching with Disaggregated Memory: High-Performance Computing in the Modern World

*Author:*
Jay Vishwarupe

*Supervisor:*
Dr. Linh Thi Xuan Phan

*A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Applied Science*

*in the*

School of Engineering and Applied Sciences (SEAS)

May 14, 2024

UNIVERSITY OF PENNSYLVANIA

# *Abstract*

Dr. Linh Thi Xuan Phan
School of Engineering and Applied Sciences (SEAS)

Bachelor of Applied Science

**Distributed Caching with Disaggregated Memory: High-Performance Computing in the Modern World**

by Jay Vishwarupe

Distributed caching has become a go-to tool for developers seeking to build efficient, scalable, and responsive systems. Popular industry frameworks have been built on traditional shared-nothing architecture to provide an appropriate balance between performance and resiliency. The inherent problem in this design is that it forces compute and memory to be coupled, requiring significant over-provisioning for peak load and driving costs up exponentially. Machine learning has only made this problem harder, as workloads are now often terabytes in size. This thesis examines how the emerging paradigm of memory disaggregation can offer a solution to these problems. Memory disaggregation refers to an architecture in which compute and memory are decoupled and are instead interconnected by a high-performance, low-latency network fabric. We examine the development of physical disaggregation (CXL) and logical disaggregation (RDMA). We develop an understanding of the eviction policies at the heart of caching, from simple primitives to intricate combinations. By grasping these principles, we can understand how to apply them to disaggregated memory systems, with existing infrastructure and next-generation platforms. Our investigation finds that while open challenges relating to performance isolation, memory allocation, and fault tolerance persist, distributed caching in disaggregated memory significantly outperforms its shared-nothing counterparts and provides a framework to grapple with the increasingly large workloads of tomorrow.

# Contents

# Chapter 1

# Introduction

With an increasingly digitized world comes more data than ever before. Over the last three decades, the monthly volume of internet traffic has grown exponentially, reaching 456 Exabytes per month in 2024 [2]. As applications scale to serve billions of users and their traffic, there becomes an increasing need for quick data access and retrieval. Concurrent with this transformation have also been prolific advancements in data processing. In particular, the advent of novel machine learning and artificial intelligence techniques that employ complex, long-running algorithms has strained what we can achieve with current computers. Given the complexity of these algorithms, it has become untenable to reproduce or recreate outputs every time they are needed when dealing with the demands of efficiency, scalability, and latency of modern applications.

One approach to mitigate these problems has been caching. Caching is a technique that involves creating a copy of computationally expensive or frequently utilized data. From CPUs to large-scale content distribution networks, caching manifests itself at every stage of the technology stack. However, traditional single-node caching, while effective, has limited scalability. Modern applications require thousands of servers spread across the globe to serve requests in a timely and efficient manner. Despite consistency and fault tolerance challenges, distributed environments offer significant performance gains. Each approach to caching in distributed networks comes with unique performance profiles and trade-offs. As such, dozens of distributed caching frameworks like Redis [39], Memcached [32], and Hazelcast [17] have been developed, each with their unique design philosophies. Such systems have proliferated throughout the industry and have become synonymous with performance in modern computing architectures.

## 1.1 Key Definitions

We introduce a few key definitions relevant to the problem space.

1. Cache: A temporary storage intended for data access and operation.
2. Object: A single entry in a cache.
3. Page: A fixed-unit entry in a cache (often used interchangeably with object).
4. Working set: The set of objects used by a workload.
5. Distributed Systems: A collection of independent servers coordinating to achieve a common goal.
6. Distributed Cache: A collection of independent servers connected by a network that pool resources to operate as a cache.

7. Hotness/Coldness: Characterization of the rate at which an object is accessed. A hot object is frequently accessed while a cold one is not.
8. Consistency: "The system behaves as if there is a single copy of the data, and all operations are executed one after another without any concurrent or out-of-order effects." [3]
9. Shared-nothing architecture: A distributed computing architecture where each node manages and owns its resources and only communicates with one another through messages.

### 1.1.1 The Memory Hierarchy

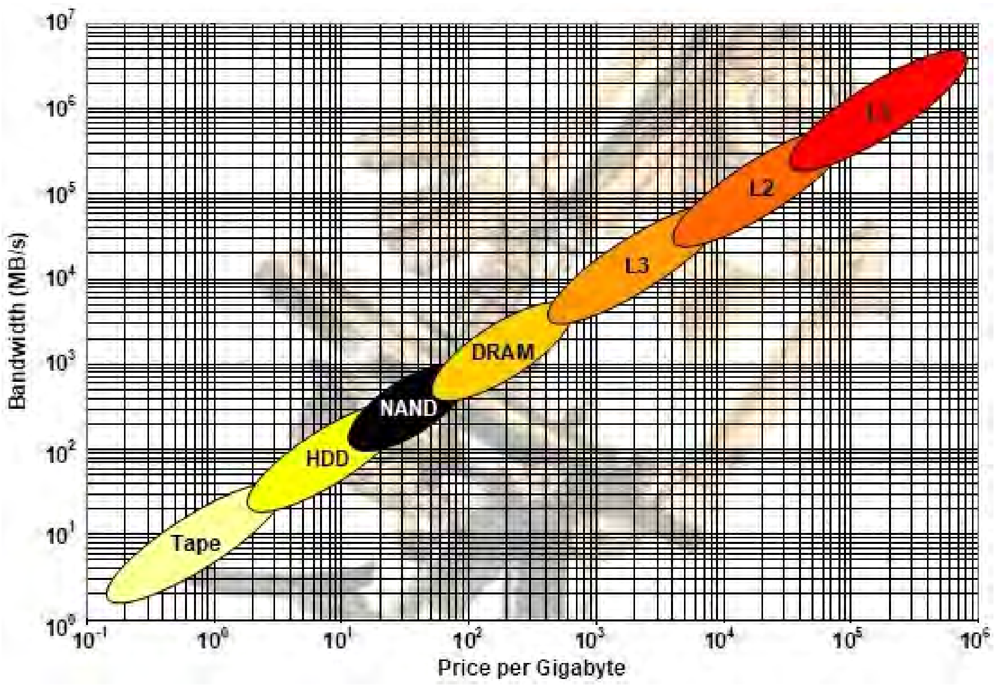


FIGURE 1: The Memory Hierarchy [15]

The memory hierarchy pictured in Figure 1 defines the storage tiers. As we go from the bottom left corner to the top right, the trade-off is increasing bandwidth at the expense of increased price per gigabyte. The speed comes from the proximity of the memory source to the compute source. The increased cost comes from two main sources:

1. **R&D costs**: Manufacturing and development costs to build increasingly faster and smaller memory multiply. For example, CPU-level caches require far more transistors per byte of storage than DRAM.
2. **Physical constraints**: Physical constraints limit the maximum speed of certain types of memory. For example, growing the CPU level caches (e.g. L1 cache) in size results in greater cache latency as it takes more type to index and sift through objects. Additionally, the limited bandwidth of communication channels can constrain the capacity of a storage level. For example, the size of

DRAM is physically constrained by the number of addressing lines connecting it to the CPU. This becomes particularly relevant as servers reach a point where they can no longer utilize more RAM, regardless of the cost.

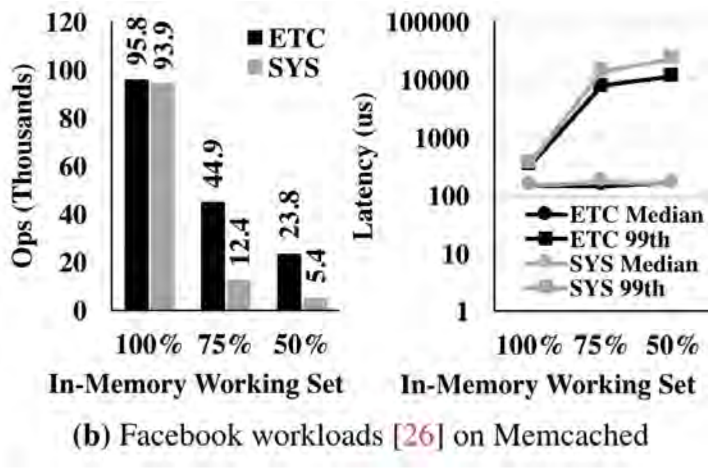


FIGURE 2: Performance of two FB workloads with varying amounts of the working set fitting in a MemCached instance [14]

Even with this limitation, the most popular distributed caches utilize DRAM. This is because the working set of most applications falls into the rough order of magnitude of 1 GB to 1 TB. However, they sometimes have to drop into Flash memory (NAND) due to the scale of their data [8]. RAM and NAND offer a happy balance between the capacity and speed required to serve the needs of modern computing applications.

The importance of an application's working set all fitting in memory is evident in Figure 2. The diagram details the performance fall-off in operations per second and latency at different levels of the working set that fits in memory for a specific set of Facebook workloads. More generally, the study found an 8× to 25× performance decrease across all their tests if half the working set doesn't fit in memory [14]. The intuition comes from the fact that the time it takes to carry out operations far exceeds the time it takes to load the relevant data. In particular, a modern CPU operates at more than 3 GHz [10]. Combined with the fact that modern CPUs can often conduct more than one operation per cycle and have multiple cores working in tandem, CPUs can execute tens of billions of instructions every second. As a result, the time to access memory or disk almost always dwarfs the time to perform any complex operation. Thus, if an operation requires even one object not located in memory, the time to retrieve it will bottleneck the system. This makes caching frequently accessed information hypercritical.

## 1.2 Caching Applications

Distributed caching has a wide array of use cases. We highlight a few of the most popular applications of distributed caching as evidence of its importance.

*1) Application Caches*: One of the most diverse and popular applications of distributed caches is at the application cache level. From social media, and e-commerce to video streaming, gaming, and mobile, almost any application at scale in the modern world utilizes this technology. Some popular companies have gone as far as to develop specialized caches tuned to their needs (e.g. Facebook's CacheLib [8] and Twitter's Twemcache [50]). Before Facebook developed CacheLib, by 2013, Facebook was already running the world's largest installation of MemCached [37]. The importance of caching comes from the demand to handle the latency and throughput requirements, mitigate database load, and enable modular scaling of persistent and non-persistent storage tiers. A frequent use-case for data stored in these caches is to tune and train online machine learning algorithms deployed in search, recommendations, and other domains.

*2) Content Distribution Networks (CDNs)*: Content distribution networks form the backbone of the web's infrastructure. They comprise an extensive network of proxy servers that serve web pages, images, videos, and more from origin servers [1]. CDNs are sometimes called edge stores because they live at the boundaries of server networks and are intended to serve data to clients geographically nearby, dramatically reducing page load times and alleviating the load on origin servers. Together, they effectively operate as a large system of distributed caches storing frequently accessed media close to users around the globe.

*3) Search Result Caches*: With over 8.5 billion search results daily and 65 billion web pages indexed [40], it's no wonder that caching is central to search at Google. Caching is especially useful here as the most popular queries appear extremely frequently. Three months of AOL search logs from 2006 revealed that just the top 100,000 queries of the 36,000,000 unique queries make up 34% of the unique queries [48]. It's no wonder that the companies that have invested the most into open-source caches also have extensive search systems.

## 1.3 Traditional Distributed Caching Architecture

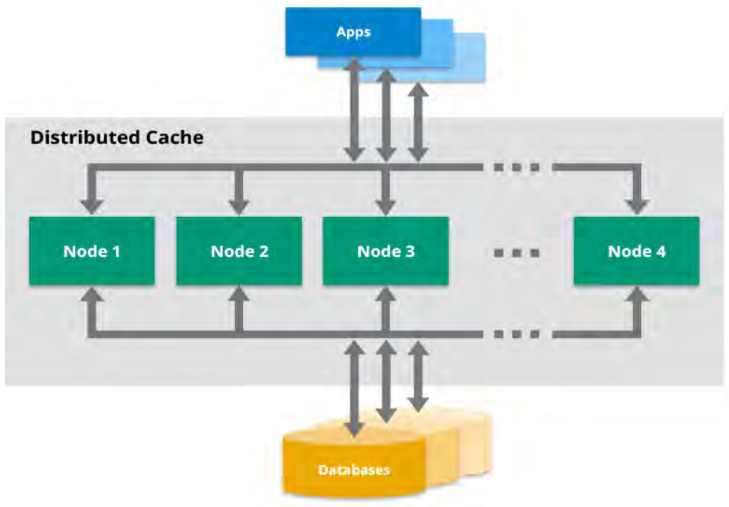


FIGURE 3: Architecture of a simple distributed cache [17]

A traditional distributed cache is composed of a series of nodes that sit between an application and an underlying data store as pictured in Figure 3. Each node uses an associative array or hash table to map hashed keys to their respective values.

Data is usually split across the nodes using a mechanism called sharding. Common ways to shard the data include modulo (send every Xth row to a node), range (split the key range into contiguous subsections assigned to nodes), and hash (assigns rows randomly to nodes via a hash function). The process equitably splits the load across the nodes so no one node becomes a bottleneck.

Data is replicated using either a primary-backup or a quorum scheme. A primary-backup architecture means a singular primary node coordinates the nodes in its shard group. The backups hold copies of the data and changes are synchronized with the other nodes synchronously or asynchronously depending on the level of consistency and performance required. A quorum approach requires that multiple nodes be queried for a given operation to ensure data is propagated to copies. The number of nodes involved in an operation is configurable based on whether the workload of the application is read or write-heavy. While primary-backup schemes are usually easier to implement and offer lower latency, quorums can offer the strongest consistency.

### 1.3.1 Caching Strategies

There are five popular caching strategies [17] used by applications depending on their needs:

1. **Cache-aside**: The application queries the cache for the data. On a hit, the entry is retrieved. On a miss, the application queries the underlying data store, updates the cache, and returns the entry.
2. **Read-through**: Identical to cache-aside except during a miss, the cache requests the data to the data store directly and updates its local entry.
3. **Write-through**: When an application updates the data store, it also updates the cache.
4. **Write-back**: The application writes data to the cache only, which asynchronously updates the data store.
5. **Refresh-ahead**: The cache is proactively populated with cache entries based on the expected usage patterns of the application.

Which strategy is chosen depends on the characteristics of the workload. For example, a workload where performance outweighs consistency would be an ideal candidate for a write-back strategy as writing to the cache is much faster than to the underlying data store.

### 1.3.2 A Novel Approach

The traditional design for distributed caches associated with modern workloads has a few fundamental issues.

*1) Limited maximum capacity*: CPUs are constrained in how much memory they can manage at once by the limited number of DDR channels and memory slots per channel [33]. This means that not only is memory expensive, but even after a certain point, it cannot be expanded any further (§ 1.1.1). This is especially problematic for the increasingly large workloads of modern applications like machine learning where the data and model sizes together can exceed a terabyte. Often intermediate results in these computations need to be stored in a distributed cache of a size proportional to the application data.

*2) Effective resource utilization*: While shared-nothing architectures provide better fault tolerance due to the isolation of each node, they also ensure that compute and memory are tightly coupled. Every node has to have a CPU and some memory pool and the addition/removal of a node necessarily means a change in both compute and memory. This is especially inefficient in distributed caching environments where memory and compute needs are not proportional (memory >>> compute).

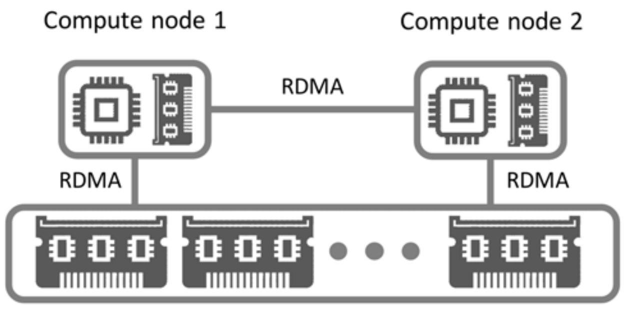


FIGURE 4: RDMA-based disaggregated memory configuration [46]

The proposed solution to both problems is to utilize a technique known as disaggregated memory. The idea of disaggregated memory is to have independent compute and memory nodes interconnected by a high-bandwidth, low-latency network fabric. Figure 4 outlines an example of an RDMA-based configuration of disaggregated memory. Not only does this dramatically increase the maximum capacity of a single server/cache system, but it also means compute and memory can be scaled independently by selecting the appropriate type of nodes to be plugged into the fabric.

Naturally, such a paradigm shift has cascading effects on how the rest of distributed caching functions. In this thesis, we will survey the development of techniques for distributed caches in disaggregated memory environments. In particular, we will first understand the types of eviction algorithms used by caches. Then, we will explore how these algorithms are applied in novel distributed caching approaches for disaggregated environments.

# Chapter 2

# Eviction Algorithms

While caches are quite fast, they trade off in capacity. Their small size makes carefully selecting what pages to retain and what pages to remove hypercritical to their performance. The algorithms that define these rules are called eviction policies. These approaches have evolved from simple beginnings to sophisticated designs that leverage modern machine learning advances.

## 2.1 Key Metrics

1. Cache Hit Ratio: The cache hit ratio is mathematically defined as:

   $$\frac{\text{Cache Hits}}{\text{Cache Hits} + \text{Cache Misses}} \times 100\%$$

   This is a common measure of how well an eviction policy performs.

2. Cache Miss ratio: The cache miss ratio is mathematically defined as:

   $$\frac{\text{Cache Misses}}{\text{Cache Hits} + \text{Cache Misses}} \times 100\%$$

   This measure of eviction policy performance is simply a mathematical restatement of the cache hit ratio. That is, Cache Miss Ratio $= 1 -$ Cache Hit Ratio.

While algorithms attempt to maximize (minimize) the cache hit (miss) ratio, they must also do this efficiently to maximize throughput and minimize the overhead of each operation. While there is no single, standardized measure of this concept due to the varying ways to measure it (under different workloads, different scales, etc.), it is undoubtedly a significant consideration when evaluating eviction policies.

## 2.2 Eviction Primitives

The most popular, straightforward approaches to eviction are known as eviction primitives. We've identified and described a sampling of the most popular primitives.

### 2.2.1 First-in, First-out

First-in, first-out (FIFO) is perhaps the most intuitive approach to eviction. Items are evicted from the cache in the same order they are inserted. Historical statistical analyses have shown approaches like least recently used (LRU) to be strictly better than FIFO, hence its unpopularity. What it lacks in performance in cache hits, it makes up for in its throughput and scalability due to its space and compute efficiency. This is because the approach requires minimal additional metadata and overhead to track what object needs to be evicted next.

### 2.2.2 Least Recently Used

Temporal locality has been a fundamental principle guiding the development of caching algorithms over the last half-century. The idea conveys that recently used pages are more likely to be used again and should be more likely to be preserved. Approaches proposed back in the 60's like Belady's MIN algorithm [7] utilize this finding in their designs.

Naturally, this gives rise to least recently used (LRU) eviction, in which the least recently used page is the first to be evicted. As items continue to be accessed, their recency is updated accordingly. The relatively simple design but high performance has led to utilization by the most popular caching frameworks [17, 32, 39].

Research has shown LRU performs better than naive FIFO in many experiments [11] due to this exact phenomenon. Such capability has made it the object of study for decades. However, the large overhead required to eagerly promote pages to the head of the queue whenever they are accessed has led to much research on how to replicate, approximate, and extend LRU's behavior (see CLOCK/CLOCK-PRO [19], Fixed Segmented LRU [35], etc.).

### 2.2.3 Least Frequently Used

Least frequently used (LFU) operates orthogonally to least recently used. Pages in the cache are given frequency counters that are incremented every time they are requested. At eviction, the page with the lowest frequency score is evicted. Maintaining, updating, and re-ranking such information for every page hit is naturally quite costly. While constant time approaches have been developed for the LFU eviction scheme [30], it's rarely chosen over LRU due to its inability to adapt to changes in the working set of an application (e.g. when a set of new pages should replace the old ones, LFU tends to hold onto the existing pages). However, certain datasets like networking do appear to exhibit better performance with LFU over LRU [25].

## 2.3 Combined Approaches

While many of the eviction primitives discussed often produce respectable performance, much of their power is unlocked when they are combined and composed in novel ways. Below, we take a sampling of different approaches taken to weaving eviction primitives together.

### 2.3.1 Adaptive Replacement Caching

LRU and LFU have unique merits that perform well under different environments. This naturally led to exploring ways to leverage both approaches and dynamically adapt to evolving requirements. This area of research yielded several popular approaches that could be characterized into two groups: offline and online. Offline approaches entail those with pre-configured parameters that determine the mix of LRU and LFU behavior exhibited during eviction. Given the dynamic nature of workloads, tuning these parameters is often complicated and insufficient. Some popular approaches in this space include LRU-2 [38], 2Q [21], LRFU [23], and MIN [7]. The second class of algorithms, online replacement, tweaks its LRU-LFU balance as it processes requests rather than being confined to static values.

While online algorithms struggle to beat their offline counterparts under ideal circumstances, they perform better under empirical ones because offline replacement cannot grapple with heterogeneous and changing workloads (e.g. fluctuating hotspots) with a static set of parameters. As the quest for the best caching algorithm waged on, one particular online algorithm, proposed by IBM research in 2003, emerged superior to its counterparts [31]. Adaptive Replacement Caching (ARC) offered a relatively simple algorithm that excelled under diverse workloads all while requiring only constant time for processing requests.

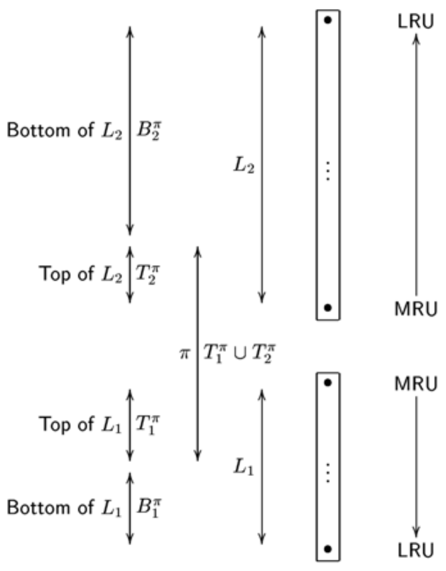


FIGURE 5: ARC algorithm design [31]

Figure 5 demonstrates the design of ARC. For caching policy $\pi$, two lists, $L_1$ and $L_2$ are maintained. $L_1$ captures pages that have been seen only once in the cache (LRU) and $L_2$ captures cache pages that have been seen at least twice (LFU). The cache has size $c$ but tracks $2c$ pages. At any given time, the cache holds a prefix of $L_1$ ($T_1$) and a prefix of $L_2$ ($T_2$). The sizes of $T_1$ and $T_2$ are governed by parameter $p$ where $|T_1| = p$ and $|T_2| = n - c$. A page is moved to either $T_1$ or $T_2$'s MRU position on a cache hit. On a cache-miss, the LRU page is placed at the MRU position of the appropriate queue $L_1$ or $L_2$.

The parameter p dynamically adapts to the current workload. If the missed page is in $B_1$ of $L_1$, we increase $p$ as LRU better captures the workload; otherwise, if it's in $L_2$, we decrease $p$. The rate at which $p$ is scaled is a function of how big the other queue is relative to the one to be grown. Intuitively, this means that when the system heavily favors one type of replacement policy but it receives requests more in line with the other, it transitions faster.

The algorithm exhibits a multitude of advantages over its counterparts:

1. **Scan Resistance**: When the cache is flooded with requests that scan (retrieve every page) once, they filter through the $L_1$ cache with minimal impact on $L_2$ meaning frequently used pages are retained.

2. **Constant Time Replacement**: The relatively simple design and implementation enables $\mathcal{O}(1)$ time replacements.

3. **High Hit Rates**: The algorithm consistently exhibits excellent empirical performance (hit ratios) when pitted against LRU in various settings as seen in table 1. In particular, ARC performs relatively best when cache sizes are small.

| Workload | c | space (MB) | LRU | ARC |
|---|---|---|---|---|
| P1 | 32768 | 16 | 16.55 | 28.26 |
| P2 | 32768 | 16 | 18.47 | 27.38 |
| P3 | 32768 | 16 | 3.57 | 17.12 |
| P4 | 32768 | 16 | 5.24 | 11.24 |
| P5 | 32768 | 16 | 6.73 | 14.27 |
| P6 | 32768 | 16 | 4.24 | 23.84 |
| P7 | 32768 | 16 | 3.45 | 13.77 |
| P8 | 32768 | 16 | 17.18 | 27.51 |
| P9 | 32768 | 16 | 8.28 | 19.73 |
| P10 | 32768 | 16 | 2.48 | 9.46 |
| P11 | 32768 | 16 | 20.92 | 26.48 |
| P12 | 32768 | 16 | 8.93 | 15.94 |
| P13 | 32768 | 16 | 7.83 | 16.60 |
| P14 | 32768 | 16 | 15.73 | 20.52 |
| ConCat | 32768 | 16 | 14.38 | 21.67 |
| Merge(P) | 262144 | 128 | 38.05 | 39.91 |
| DS1 | 2097152 | 1024 | 11.65 | 22.52 |
| SPC1 | 1048576 | 4096 | 9.19 | 20.00 |
| S1 | 524288 | 2048 | 23.71 | 33.43 |
| S2 | 524288 | 2048 | 25.91 | 40.68 |
| S3 | 524288 | 2048 | 25.26 | 40.44 |
| Merge(S) | 1048576 | 4096 | 27.62 | 40.44 |

TABLE 1: A comparison of hit ratios for the ARC and LRU replacement policies across varying workloads and cache sizes [31]

While this framework is quite flexible, it does struggle in one particular case. There is constant churn when the working set of pages is slightly larger than the cache size. This is because, in this pattern, the page that is just about to be used has just been evicted. Such behavior repeats, resulting in continuous churn and high overhead. While incremental improvements on the ARC framework have been proposed like mARC [42] to deal with this phenomenon known as the ARChilles' heel, they target a slightly different subset of the caching problem space in which new pages do not always have to be added to the cache.

### 2.3.2 Learning Cache Replacement

The principles of the ARC have been lifted and translated into the machine-learning world. In particular, this combined approach can be recharacterized as a multi-arm bandit problem from reinforcement learning and solved using online-learning-based regret minimization techniques.

Reinforcement learning (RL) is one of three key machine learning paradigms. It deals with problems of dynamic environments in which cumulative reward must be maximized. However, the exploitation-exploration trade-off remains a key challenge of solutions addressing this class of problems. Exploration refers to the process of discovering new actions an agent can take. Exploitation is the process of an agent utilizing an action from its knowledge set that will maximize its immediate reward. The tricky challenge is to balance between exploring to uncover the actions that may maximize reward and exploiting the current actions to build up cumulative reward.

Perhaps the space where this exploration-exploitation trade-off has been studied the most is in the space of the multi-arm bandit problem, a subclass of RL. One can think of this problem as an agent who must operate a series of $N$ slot machines, each with independent probability distributions governing their reward/win rates. An agent is tasked with maximizing the reward they win and must balance between exploring slot machines to gauge their reward rates and exploiting known slot machines to play the ones it knows will maximize its immediate reward.

This multi-arm bandit problem can be applied to cache eviction by considering LRU and LFU as the two arms of a bandit. During an eviction, the model must decide which policy to apply to maximize its reward. Only two caching policies are chosen to minimize the time the model spends in exploration. Additionally, LRU and LFU are chosen because the policies are orthogonal in their philosophy of eviction and produce vastly different results. Here, the reward is based on regret minimization of making an eviction decision. Every eviction is graded based on how good or bad of a decision it was, and the agent is tasked with minimizing this regret.

The **Le**arning **Ca**che **R**eplacement (LeCaR) approach utilizes this multi-arm bandit characterization with the additional assumption that a dynamic combination of LRU and LFU can handle any workload. In particular, the task of LeCaR is to tune and adapt weights for a probability distribution that determines which eviction policy will be used.

The model maintains a FIFO queue of the history of evictions by LRU and LFU denoted $H_{\mathbf{LRU}}$ and $H_{\mathbf{LFU}}$ respectively. The history is sized equally to the cache itself. Evictions of entries in history are graded based on whether the entry being maintained in the cache would have prevented a cache miss. Intuitively, more recently evicted entries are punished more as it reflects the deliberate decision to eliminate something used recently, a potentially risky decision.

*Algorithm 1*: The algorithm responds with the relevant page if it is present, updating the metadata for the cache with it. If it is not present, it removes the page from the history of $H_{\mathbf{LRU}}$ or $H_{\mathbf{LFU}}$ since we are about to re-insert the page in the cache. The weights are updated according to 2. Then, if eviction is necessary, it samples the Bernoulli distribution to determine the appropriate eviction policy. It then removes the last page from its corresponding history FIFO queue if it is full, adds the evicted page to the head of the history queue, and then removes the old page from the cache. Finally, the new page is added to the cache.

*Algorithm 2*: The weights are updated whenever we encounter a cache miss. We identify the eviction policy responsible for the eviction of this page (e.g. the mistake we "regret"). Intuitively, we want to increase the weight on the counterpart eviction policy since it is more optimal. This is done using an exponential quantity and is scaled based on how bad the eviction was (e.g. a more recent eviction is punished more). Finally, the weights are re-normalized.

The LeCaR model consistently outperforms LFU, LRU, and the combined ARC approach. In Figure 6, the algorithm exhibits the highest hit rate. The tested dataset utilized changing behavior in workload and the model adapted quickly, as evident in the re-weighting between LRU and LFU. In real-world datasets, the model exhibits a hit rate up to 18x of ARC at the cost of double the overhead. The model more generally performs relatively better when the cache size is smaller. This is because it is in these exact situations that the working set doesn't fit completely in the

**Algorithm 1** LeCaR(LRU,LFU)

```
INPUT: requested page $q$
if $q$ is in $C$ then
    $C$.UPDATEDATASTRUCTURE($q$)
else
    if $q$ is in $H_{LRU}$ then
        $H_{LRU}$.DELETE($q$)
    else if $q$ is in $H_{LFU}$ then
        $H_{LFU}$.DELETE($q$)
    end if
    UPDATEWEIGHT($q, \lambda, d$)
    if Cache $C$ is full then
        ACTION $\leftarrow$ (LRU, LFU) with probability $(w_{LRU}, w_{LFU})$
        if ACTION = LRU then
            if $H_{LRU}$ is full then
                $H_{LRU}$.DELETE(LRU($H_{LRU}$))
            end if
            $H_{LRU}$.ADD(LRU($C$))
            $C$.DELETE(LRU($C$))
        else
            if $H_{LFU}$ is full then
                $H_{LFU}$.DELETE(LRU($H_{LFU}$))
            end if
            $H_{LFU}$.ADD(LFU($C$))
            $C$.DELETE(LFU($C$))
        end if
    end if
    $C$.ADD($q$)
end if
```

**Algorithm 2** UPDATEWEIGHT(q, $\lambda$, d)

```
Require: page q, learning rate $\lambda$, discount rate d
    $t :=$ time spent by page $q$ in History
    $r := d^t$
    if q is in $H_{LRU}$ then
        $w_{LFU} := w_{LFU} \cdot e^{\lambda \cdot r}$                ▷ increase $w_{LFU}$
    else if q is in $H_{LFU}$ then
        $w_{LRU} := w_{LRU} \cdot e^{\lambda \cdot r}$                ▷ increase $w_{LRU}$
    end if
    $w_{LRU} := \frac{w_{LRU}}{w_{LRU}+w_{LFU}}$                     ▷ normalize
    $w_{LFU} := 1 - w_{LRU}$
```

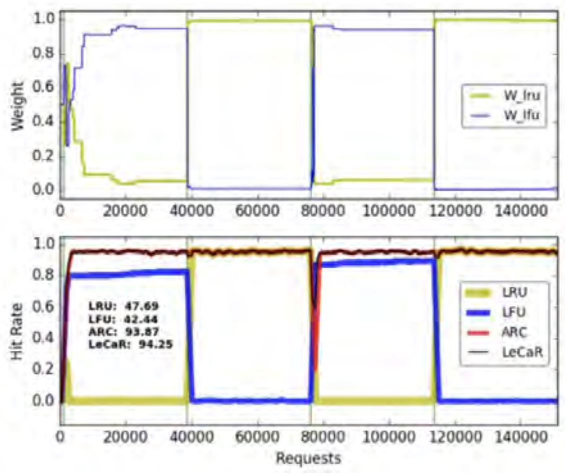


FIGURE 6: Dynamic performance of LeCaR

cache. Hence, evictions are frequent, and judicious decisions are rewarded highly with better hit rates.

Since its creation, researchers have found ways to improve the algorithm further. A modified version of the algorithm called OLeCaR [53] utilizes a new adaptive reinforcement learning algorithm EXP4-DFDC to dynamically set an optimal learning rate rather than use a static, empirical value as done in the LeCaR paper.

Thus, while ARC's heuristic and other approaches offer simpler and faster implementations, the learning-based strategy reflects one of the first ML-based techniques that shows a marked improvement over its deterministic counterparts.

### 2.3.3 S3-FIFO

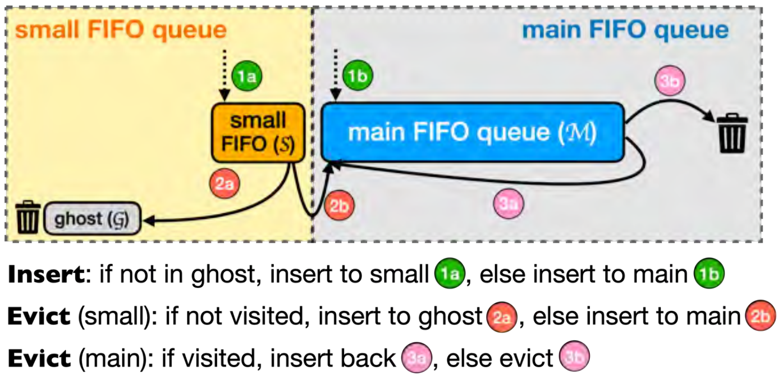
While recent approaches have become increasingly complicated and required greater overhead, breakthroughs in the space have yielded a return to fundamentals and simplicity. Two key traits make eviction algorithms effective: evicting unpopular objects fast (quick demotion) and efficiently retaining popular objects (lazy promotion) [51].

*1) Quick Demotion*: The most common culprit for unpopular pages to be evicted are one-hit wonder pages which are only ever requested once (often common in scan). These objects must be evicted as soon as possible to prevent cache pollution. Findings have shown that most contemporary algorithms wait too long to evict objects as they try to determine if an object should be retained or not. When applying a probationary FIFO queue on even existing algorithms like ARC and LIRS [20], their miss ratios were improved by up to 59.8% and 49.8% respectively.

*2) Lazy Promotion*: While most algorithms proactively re-order or re-rank entries on cache hits (eager promotion), lazy promotions posits that this promotion process should only be done at the point of eviction. The primary benefit comes from improvements in throughput as the amount of work and metadata managed on any given cache hit is significantly reduced. Secondarily, better promotion decisions can be made at eviction time as more insight can be accumulated about the object while it traverses through the cache. When tested on 5307 production traces from 10 data sources, variants of lazy promotion FIFO regularly exhibited lower miss ratios than LRU.

These principles culminated in the development of a contemporary approach called S3-FIFO. The approach utilizes just three static FIFO queues demonstrated empirically lower miss ratios than its counterparts in 6594 cache traces [52].

As seen in Figure 7, an intermediate filtering queue, known as the small FIFO, filters out objects only accessed once. The items evicted from the small FIFO queue are put into a ghost queue. This ghost queue potentially promotes objects into the main FIFO queue based on whether the object was visited.



**Insert**: if not in ghost, insert to small 1a, else insert to main 1b

**Evict** (small): if not visited, insert to ghost 2a, else insert to main 2b

**Evict** (main): if visited, insert back 3a, else evict 3b

FIGURE 7: S3-FIFO design [49]

The success of this approach is a product of its adherence to the principles of lazy promotion and quick deletion. Objects are only (re-)promoted to the main queue when evicted from the small or main FIFO queues. Objects also undergo prompt removal due to the small nature of the FIFO queue (generally about 10% of the total cache size).

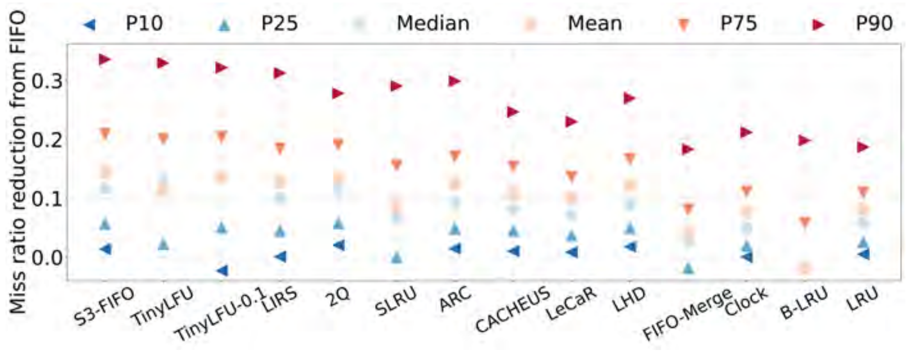


FIGURE 8: Improvement in miss ratio of S3-FIFO as compared to other leading eviction policies (higher is better) [49].

Careful examination of many real-world traces shows that they truly exhibit the one-hit-wonder phenomenon S3-FIFO can benefit from. This yields the following benefits:

*Scalability*: FIFO queue implementations are quite efficient and require only constant overhead with respect to the queue. In particular, no additional metadata like frequency must be tracked alongside the object entry. FIFO also avoids the need to lock when reading or writing, increasing its throughput as compared to its counterparts like LRU which must do more complicated book-keeping.

*Performance*: Analysis finds that algorithms with faster and precise demotion speed of entries sporadically accessed tend to perform best on real-world data [52].

S3-FIFO robustly guarantees that unpopular objects are evicted timelier than any of their counterparts. As seen in Figure 8, when compared with a standard FIFO eviction policy and a variety of other popular replacement policies, S3-FIFO exhibits the greatest improvement in miss ratio. An analysis of 153 in-memory cache clusters with over 80 TB of data similarly concluded that even vanilla FIFO is best suited for production cache replacement strategies [50].

Similar approaches like SIEVE [54] have been developed using the same principles to beat LRU although its downfall is its lack of scan resistance. Shockingly, the simple composition of FIFO primitives significantly outperforms the increasingly complicated eviction approaches developed during the last twenty years. This leaves us with the conclusion that sometimes, less is more.

# Chapter 3

# Memory Disaggregation

## 3.1 Background

Modern internet-based applications have become increasingly data-intensive. To serve millions of clients and billions of requests, applications have relied on distributed memory-based caches.

The fundamental challenge with memory-based caches in contemporary distributed systems is that they don't adapt well to the heterogeneous load of web services. This high volatility in activity means administrators must provision for peak load, leaving much of the memory idle at all times. For example, one deep-dive into traces of Alibaba's infrastructure found that memory utilization sat below 50% on average [27]. In some cases, 50% of the server operating costs come from DRAM alone [36], much of which is devoted to distributed caches. Given that the compute and memory needs are often uncorrelated, the natural question arises of how to separate them so they can be provisioned accordingly. This exact question has given rise to one of the most prolific domains for research in recent years: disaggregated memory.

### 3.1.1 Memory Disaggregation

Memory disaggregation is the process of decoupling compute from memory. This concept of disaggregation itself is not novel in computer science. Perhaps the most canonical example comes in the transition from large, monolithic mainframe computers to distributed systems as we know them today. Memory disaggregation follows similarly and can be implemented at the hardware and software levels as seen in Figure 9.

**Hardware**: Compute and memory are completely physically separated. Separate compute and memory blades are created with the first containing only CPUs with minimal memory and the second containing minimal CPUs with mostly memory. These blades communicate with one another with a high-bandwidth network.

**Software**: Traditional compute-memory coupled hardware is transparently used. Computers utilize an appropriate portion of their local memory. Excess memory is "leased" out to the broader cluster and repurposed as part of the memory address space of a remote process. This result is a single process's memory space mapping to dozens of computers underneath the hood. Traditional underlying hardware in this case is utilized transparently.

This exact kind of memory disaggregation has also been around for decades. The earliest implementations of this system were known as non-uniform memory access (NUMA) [12] and were designed for high-performance compute applications. NUMA utilizes physically separated memory with high-bandwidth interconnects

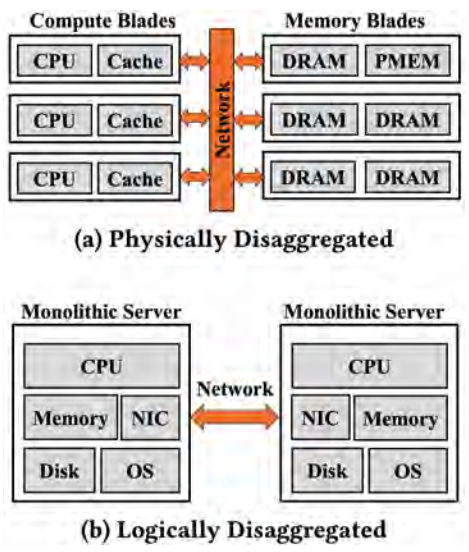


FIGURE 9: Physical vs Logical Disaggregation [29]

so that its processors can fetch data from anywhere in the system. While the technology appeared promising, it was dwarfed by its much more popular counterpart, "shared-nothing" where each node in a distributed system is completely independent with its own compute, memory, storage, etc., and only communicates through messages with one another. This model has become an industry standard. We can identify a few key reasons for this phenomenon:

1. **Scalability:** It's easier to insert an extra node in a system where they are independent rather than highly coupled.

2. **Accessibility:** Shared-nothing architectures can operate with commodity hardware that is highly heterogeneous, as long as they can adhere to a consistent communication protocol.

3. **Fault Tolerance:** The independent, loosely coupled component design eliminates any single point of failure.

While shared-nothing architectures are omnipresent in data centers today, there has been a greater push toward exploring novel disaggregation techniques. As we reach the asymptotic limits of Moore's laws and rules of physics, engineers are left with no choice but to turn to specialized hardware to achieve the performance they need.

In the context of distributed caching, this means redesigning systems to leverage elastic memory environments and grappling with the challenges of running eviction algorithms without single entry points.

### 3.1.2 Remote Memory Access

Remote memory access is a communication paradigm that enables a process to access the memory of another node without invoking a remote process. RMA is a programming model for memory that distinguishes between local and remote memory [18]. RMA offers a means to achieve the logical disaggregation seen in Figure 9b.

### 3.1.3 Remote Direct Memory Address

Remote direct memory address (RDMA) is a lower-level primitive that enables high-throughput, low-latency data transfer. In particular, it allows a process to access memory remotely without invoking the remote CPU. The RMA interface is often implemented with RDMA. RDMA does, however, support additional functionality beyond the RMA specification (e.g. two-sided message passing) [18]. RDMA operates on various network architectures including InfiniBand and RDMA over Converged Ethernet (RoCE) [13].

Memory frameworks built on RDMA have been created to offer page-based remote memory abstractions. Infiniswap [14] represents one of the first frameworks with transparency at the hardware, OS, and application levels. These memory systems must grapple with issues like remote memory latency minimization, performance isolation, memory heterogeneity, additional failure modes, scalability, and security [29].

### 3.1.4 Compute Express Link

Although RDMA provides logical disaggregation at the network level, a gap persisted in achieving physical disaggregation of hardware components (Figure 9) within an individual server. In 2019, the Compute Express Link (CXL) framework offered a radical rethinking of memory [36]. The technology came from a desire to solve a slightly different problem but is undoubtedly a boon to distributed caching. Computer processors internally employ specialized units like the arithmetic logic unit (ALU) and the memory management unit (MMU). Hardware accelerators generalize this principle, acting as external processors purpose-built for specific tasks that work with the rest of the computer. Normally, these hardware accelerators maintained their own pool of memory which imposed two-fold downsides:

1. Memory provisioned for each component means frequently underutilized memory sitting idle as the pools are isolated.
2. Additional data copies must be made whenever data between components (e.g. CPU and GPU) needs to be exchanged.

CXL [4], a cache-coherent open interconnect standard, provided a solution to these issues by enabling high-speed communication between hardware accelerators and other components within a server. Built on the PCIe standard, CXL technology allows hardware accelerators to access and share byte-addressable memory pools and dedicated memory expansion cards. CXL-based memory pooling platforms are already in development, with one of the most promising being Pond, a collaboration between Microsoft and Google [24]. Their research reveals that although CXL systems may experience latencies up to 222% more than DRAM, this difference is significantly smaller than the performance gap between DRAM and SSDs, which can be an order of magnitude or more.

With the advent of CXL 3.0 comes the opportunity to expand the fabric topologies supported by the interconnect. The update leverages PCIe 6.0 to double the bandwidth to 64GT/s and expands support to 4,096 nodes within a single network all without increasing latency [41]. Figure 10 demonstrates how these changes can be utilized to design a complex CXL fabric with multiple attached devices each with their own and shared attached memory.

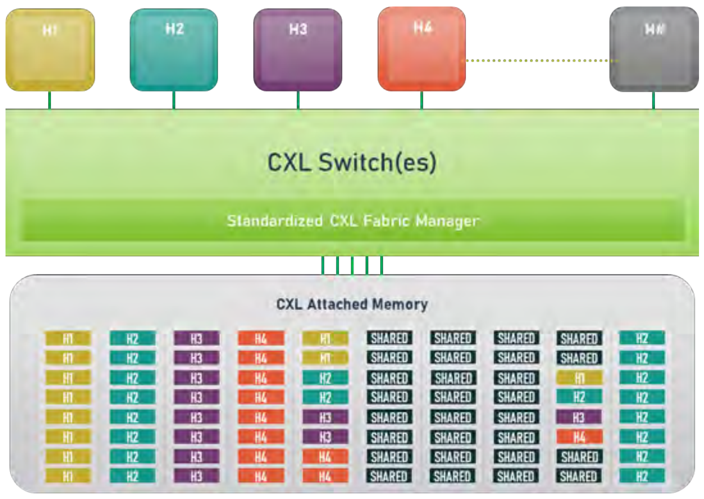


FIGURE 10: Complex topology of a possible CXL fabric [9]

### 3.1.5 A Hybrid Approach

The future of disaggregation involves combining physical and logical disaggregation. Since CXL systems are limited in the distance they can cover, RDMA can be used to interconnect CXL clusters with one another. One of the most recent of these frameworks is Rcmp [47]. The framework addresses the incongruencies between the two systems regarding performance and cache line granularity and yields 5.2× lower latency and 3.8× higher throughput than RDMA-based systems. Together, CXL and RDMA offer memory utilization in ways never seen before:

1. **Efficient Memory Utilization:** Idle memory can be re-assigned and utilized by physical servers that may encounter higher traffic. Additionally, external fragmentation can be minimized as objects/data too large to fit onto a single server can be transparently split across many different servers.

2. **Load-based Adaptation:** Data migration when nodes enter and exit the cache cluster can be complex, time-consuming, and resource-heavy during resharding events [32, 39]. The abstraction of a singular contiguous block of memory accessible to the entire compute pool means data rarely needs to be copied. Only in exceptional cases where a hot data entry is causing a performance bottleneck will data be additionally split/replicated to balance the load.

However, RDMA brings a new host of challenges as well:

1. **Increased Latency**: While the standards continue to improve, CXL introduces at least a 50-100 nanosecond increase in latency as compared to DRAM [29]. This comes from the controller, up to two switches, and a network each operation must pass through in CXL [16].

2. **More points of failure**: Singular sections of data (and potentially even objects) may now be split across many computers. Thus, the failure of one node has significantly more cascade effects than previous systems. This is as opposed to shared-nothing systems where individual nodes maintain their physical resources.

3. **Complicated eviction algorithms**: Architectures in which the CPU is bypassed mean servers have no bookkeeping on cache accesses. This requires novel client-centric eviction approaches.

The intricate tradeoffs make disaggregated memory a powerful but challenging tool that demands further research and exploration.

## 3.2 CliqueMap

While distributed caching frameworks that leverage disaggregated memory have been proposed for years, Google's CliqueMap [5] was one of the first to legitimize the concept and initiate more widespread adoption of the approach. Previous approaches like MICA [26] helped laid the foundation for efficient multi-core memory access that bypassed the kernel by implementing a custom, lightweight networking stack that would map client requests to the cores which would process requests most efficiently. Others like HERD [22] and Pilaf [34] took varying approaches to utilizing RDMA to build key-value stores. While promising, the industry appeared hesitant to deploy these prior solutions because they lacked the fault tolerance, scalability, deployment, availability, flexibility, and testing that came with their established counterparts. [32, 39].

### 3.2.1 Design & Architecture

CliqueMap synthesizes the insights and lessons from its predecessors to provide a production-caliber implementation of a remote memory access (RMA) based distributed cache, backed by a credible name organization like Google. As of 2021, CliqueMap deployments managed 1 PB of DRAM and served queries at 150M per second.

Central to the design of CliqueMap was the consideration of how to evolve distributed caching while maintaining interoperability with existing infrastructure. In essence, CliqueMap provides a roadmap for performance in a brownfield environment with potential for improvement under newer infrastructure.

CliqueMap's core design employs a hybrid usage of remote procedure calls (RPC) and remote memory access (RMA). Remote procedure calls are a mechanism that enables developers to transparently call remote procedures in distributed environments while maintaining the abstraction of it being a local call. While this technique offers ease of programmability and robust support for fault-tolerance, error-handling, and retries, it comes at the minimum cost of at least $50\mu s$/operation in overhead. In contrast, the no-frills RMA can offer a low latency and high-bandwidth pipeline along the idempotent read-only path where issues of concurrency and race conditions are minimal.

Entries in CliqueMap are stored in an associative hash table, with write requests carried out via RPCs. The normal course of reads is conducted via a pattern known as 2xR. The client retrieves the associated IndexEntry which holds the location of the key-value pair in the data region as seen in Figure 11. It then directly fetches the DataEntry from the Data Region. Both these operations are conducted via RMA, bypassing the server CPUs and offering significant speedup.

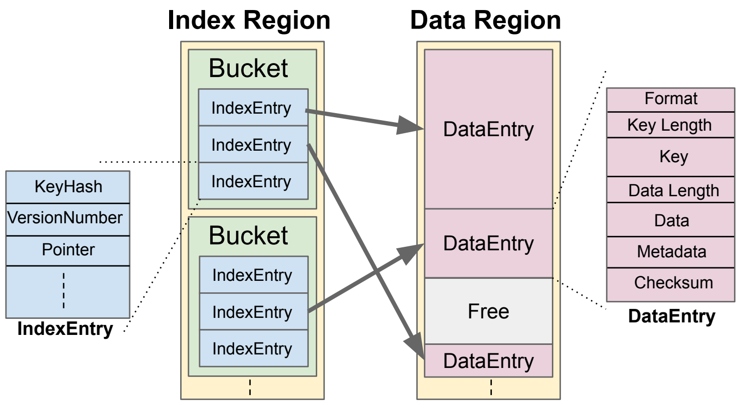


FIGURE 11: Layout of data in CliqueMap[5]

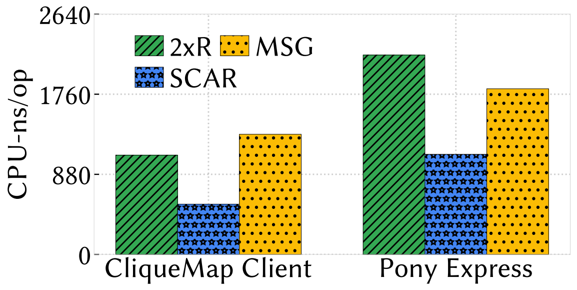


FIGURE 12: Comparison of a standard 2xR vs SCAR lookup[5]

### 3.2.2 Evaluation

The CliqueMap authors benchmarked the framework under different load settings and networking infrastructure. For example, RMA allows CliqueMap to be deployed on various transports like Pony Express [28], 1RMA [45], and RDMA. Some of these protocols even offer advanced RMA primitives like scan and read (SCAR) which can eliminate one network hop in the 2xR and effectively double the speed of operation. Figure 12 demonstrates this speedup on both the client and server sides. Standard RPCs can be used for reads and writes on deployments without support for RMA. Such flexibility makes CliqueMap an ideal solution for environments entrenched in heterogeneous data center deployments. Figure 13 shows that CliqueMap also achieves effectively constant lookup times for common-sized objects, which is ideal for read-heavy workloads.

However, an important caveat of using RMA is that server CPU bypass means only clients have a record of data access patterns. This information is necessary to conduct efficient eviction using algorithms discussed in Chapter 2. CliqueMap's solution is for clients to periodically send over batched metadata about object accesses to servers in a background process, offering timely enough information to still reasonably approximate many popular eviction algorithms like LRU (§ 2.2.2) and ARC (§ 2.3.1).

Combined with other features like transparent retries, interoperability with heterogeneous software and hardware stacks, and high availability through quorum deployments, CliqueMap cemented itself as one of the first successful disaggregated distributed caching frameworks at scale.

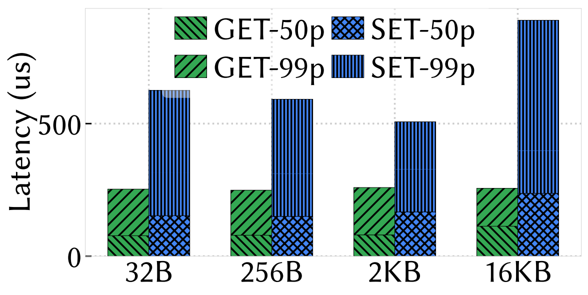


FIGURE 13: Latencies of operations on common-sized objects[5]

## 3.3 Ditto

While CliqueMap has been tailor-made to address the challenges of leveraging current infrastructure, Ditto, a state-of-the-art caching framework, presents an approach with greenfield technologies hyper-optimized for speed and performance [43]. The framework was designed to work with both RDMA and CXL systems.

The authors identify two hurdles that make distributed caching in disaggregated memory environments distinctly more challenging than its counterparts:

*1) Evaluating data hotness:* Caching in a disaggregated memory environment means there is no centralized point on the data path to keep track of data access patterns. Requests may query different memory nodes and will bypass all the CPUs. Even though clients may keep track of their access patterns, they cannot be aware of those of their neighbors. Even maintaining a separate data structure to keep track of this information would significantly bottleneck throughput due to sequential reads/writes and lock contention.

*2) Re-scaling resources:* Disaggregated memory environments offer the double-edged sword of being hyperelastic to the system demands. Changes to the fundamental characteristics of the system in terms of compute, load, and memory capacity can dramatically change what is the most efficient eviction policy, rendering traditional static approaches to caching insufficient.

### 3.3.1 Design & Architecture

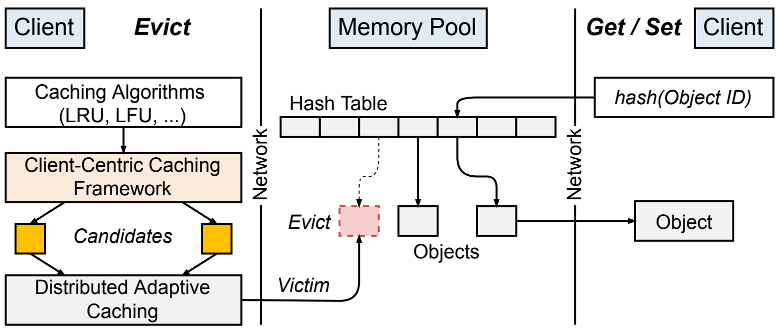


FIGURE 14: Architecture of Ditto [44]

The core design of Ditto design is pictured in Figure 14. Ditto proposes a distributed hotness measure with sample-based eviction to evaluate data hotness. Ditto co-locates and automatically updates a small set of metadata associated with each object in the memory pool that includes size, insert timestamp, last access timestamp, frequency, latency, and cost to fetch from the storage server. Metadata fields tagged as global must be stored in the object metadata itself while its local counterparts can be maintained in the client. Stateful objects which are those that depend on their prior values are grouped to be able to be updated in a single write. Such optimizations minimize the additional overhead needed to access the underlying data. To sample the candidates that may be evicted, a sample-friendly hash table allows fetching multiple contiguous objects from an arbitrary offset in a single read. Combining the breadth of metadata with the sampling mechanism allows Ditto to implement various distributed adaptive caching policies.

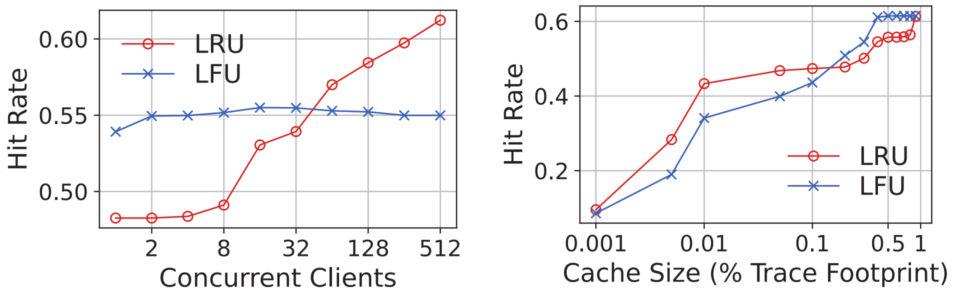


FIGURE 15: Performance of LRU and LFU under varying load and capacity sizes [44]

To address the second issue of adapting to dynamic environments, Ditto uses a variety of eviction algorithms. In Figure 15, we see the results of a simulation with varying client counts (load) and cache size (capacity). LRU (§ 2.2.2) and LFU (§ 2.2.3) perform better under different circumstances. Ditto achieves distributed adaptive caching by reformulating the problem as a multi-arm bandit problem. The authors implement a modified version of the LeCaR approach (§ 2.3.2). This was achieved with a lightweight logical FIFO queue to keep track of eviction history and a lazy weight update scheme like the original LeCaR. Combined with regret minimization to adjust which eviction algorithm is deployed, Ditto successfully achieves elasticity under changing load and capacity.

### 3.3.2 Evaluation

Ditto significantly outperforms the competition, including distributed caches designed for shared-nothing architectures like Redis [39] and even other modern RMA-based alternatives like CliqueMap [5].

As expected, Ditto defeats Redis due to the inherent challenges of shared-nothing designs. Redis shards data, meaning hot nodes in skewed workloads are bottlenecked by the single CPU that can access it. This is exacerbated by the migration costs incurred during capacity changes and the additional CPUs, often unnecessary, required for every addition of memory (due to their coupled nature). In testing, the authors found that Redis can take hundreds of seconds to rebalance and reshard while Ditto's performance effectively adjusts immediately.

Ditto was also benchmarked against CliqueMap, the current state-of-the-art, and Shard-LRU, a straightforward LRU-based implementation for disaggregated memory environments. In tests with the Yahoo Cloud Serving Benchmark [6], a standardized set of research workloads, Ditto achieves up to 9x the throughput. Figure 16 shows the p99 latency of Ditto stays about constant and only begins to creep at the

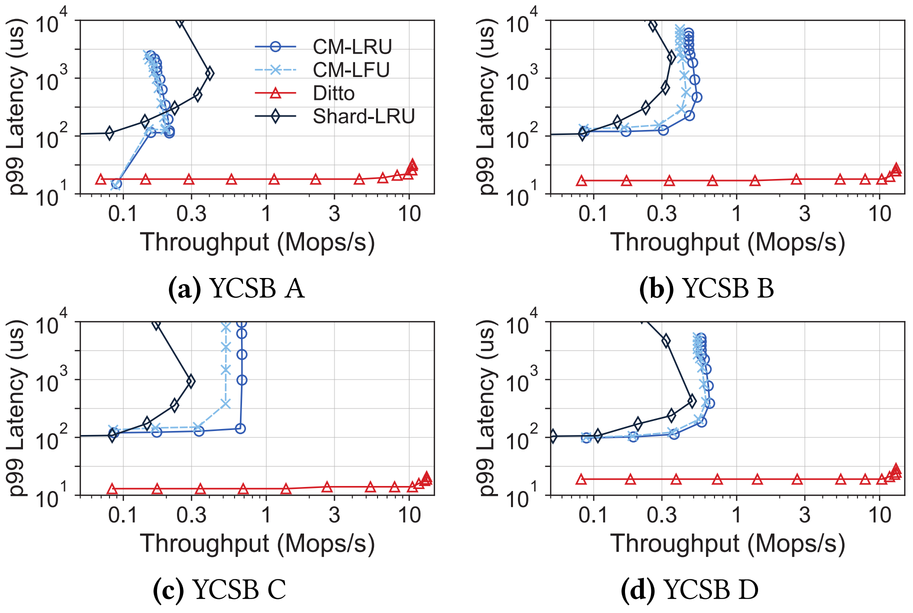


FIGURE 16: Tail latency as a function of throughput under popular workloads [44]

end when bottle-necked by the memory nodes' RDMA Network Interface Controller (RNIC).

We can identify two key reasons why Ditto outperforms Cliquemap:

*1) Ditto better executes eviction algorithms.* CliqueMap struggles under heavy-write workloads due to its use of RPCs processed by CPUs on the cache side. In contrast, Ditto's use of client-side caches means reads and writes don't require special handling that may limit throughput.

*2) Ditto better adjusts to changing environments.* CliqueMap is relegated to a single eviction algorithm at a time and can only use RPCs to send batched access information periodically to the system. This means the information is not timely and forces the system to expend compute cycles on collating this information. In contrast, Ditto's adaptive caching algorithm continuously monitors and updates the eviction policy in use with every single access that occurs.

In real-world benchmarks across five different workloads and with multiple cache sizes, Ditto consistently demonstrated superior throughput compared to CliqueMap, namely CM-LRU and CM-LFU.

Ditto fundamentally re-imagines what distributed caching means while applying advancements in eviction policies in novel ways.

# Chapter 4

# Conclusion

Reflecting on the development of distributed caching over time, we have come quite far since the days of the original NUMA approaches from the 1980s and 1990s. While traditional distributed caching has brought us far, the time is ripe for novel disaggregated memory breakthroughs that can revolutionize the field in much-needed ways.

## 4.1 Continued Areas of Research

While distributed caching in disaggregated memory has come far, open challenges that require additional innovation persist. We highlight the three most significant problems on the horizon [29]:

*1) Performance Isolation*: Although CXL 3.0-based systems support up to 4,096 nodes and numerous concurrent applications, ensuring performance isolation among them remains an open challenge. These applications contend for memory of different tiers, network bandwidth, OS scheduling time, and more. Finding the appropriate way to share finite resources will be an important hurdle to achieving scalability.

*2) Memory Heterogeneity*: An application faces many additional memory tiers in large, disaggregated memory environments. This includes but is not limited to DRAM of the running process, memory modules in the local CXL fabric, and remote memory accessible by RDMA. The challenge remains of how applications ought to balance the distribution of their data across these tiers and how this should be done in tandem with other concurrent applications.

*3) Fault Tolerance*: One of the benefits of a shared-nothing architecture is that the nodes manage and own their resources. This means that the failure of any single node is not debilitating to the entire system. However, for disaggregated approaches, one could have various issues ranging from correlated faults like failures in the CXL fabric that impact all the nodes to issues with even single memory modules used by many applications. Such problems require a rethinking of fault tolerance mechanisms best suited to this problem space.

## 4.2 Review

Overall, we've seen significant advancements along all dimensions of the field. In terms of eviction policies, we saw how simple eviction algorithms (LRU, LFU, and FIFO) have both served as effective replacement policies in a standalone manner and as primitives to be composed and combined in new and interesting ways (ARC, LeCaR, S3-FIFO). In terms of novel caching approaches in disaggregated memory, we saw how Google developed CliqueMap to serve as one of the first industrial strength caches of its kind that was highly interoperable with the heterogeneous technology infrastructure of today. Meanwhile, Ditto showcased how RDMA and CXL can be fully leveraged to maximize performance. Ditto and similar technologies represent the culmination of decades-long advancements. As researchers push the boundaries of what is possible, the future of disaggregated memory shines brightly, with even more groundbreaking developments on the horizon.

# *Acknowledgements*

I would like to thank my thesis advisor, Dr. Linh Thi Xuan Phan, for her invaluable advice and guidance in navigating the development of this paper as well as for her course CIS 5050 which sparked my passion for distributed systems. I would like to thank my thesis advisor, Professor Norm Badler, for giving valuable advice and feedback throughout the semester. Thank you to the reseachers and engineers who worked tirelessly to contribute to the advancements of disaggregated memory and distributed caching. I'd like to thank all my other peers, teachers, and professors who helped nurture my growth and development. Most of all, I'd like to thank my parents whose unwavering love, support, and sacrifices have made me who I am today.

# Bibliography

[1] 2021. URL: https://www.cdnetworks.com/web-performance-blog/cdn-benefits/.

[2] 2023. URL: https://www.ibisworld.com/us/bed/internet-traffic-volume/88089/.

[3] 2023. URL: https://www.scylladb.com/glossary/consistency-models/#:~:text=Consistency%20Models%20in%20Distributed%20Systems,-Strong%20and%20weak&text=The%20system%20behaves%20as%20if,out%2Dof%2Dorder%20effects..

[4] 2024. URL: https://www.rambus.com/blogs/compute-express-link/.

[5] Aditya Akella et al. "CliqueMap: Productionizing an RMA-Based Distributed Caching System". In: 2021.

[6] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. "YCSB and TPC-H: Big Data and Decision Support Benchmarks". In: *2014 IEEE International Congress on Big Data*. 2014, pp. 800–801. DOI: 10.1109/BigData.Congress.2014.128.

[7] L. A. Belady. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems Journal* 5.2 (1966), pp. 78–101. DOI: 10.1147/sj.52.0078.

[8] Benjamin Berg et al. "The {CacheLib} caching engine: Design and experiences at scale". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 753–768.

[9] Kurtis Bowman. *Explaining CXL memory pooling and sharing - compute express link*. 2023. URL: https://computeexpresslink.org/blog/explaining-cxl-memory-pooling-and-sharing-1049/.

[10] *CPU speed: What is CPU clock speed?* URL: https://www.intel.com/content/www/us/en/gaming/resources/cpu-clock-speed.html#:~:text=A%20CPU%20with%20a%20clock,handled%20over%20multiple%20clock%20cycles..

[11] Asit Dan and Don Towsley. "An approximate analysis of the LRU and FIFO buffer replacement schemes". In: *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '90. Univ. of Colorado, Boulder, Colorado, USA: Association for Computing Machinery, 1990, 143–152. ISBN: 0897913590. DOI: 10.1145/98457.98525. URL: https://doi.org/10.1145/98457.98525.

[12] Fabien Gaud et al. "Challenges of Memory Management on Modern NUMA System: Optimizing NUMA systems applications with Carrefour". In: *Queue* 13.8 (2015), 70–85. ISSN: 1542-7730. DOI: 10.1145/2838344.2852078. URL: https://doi.org/10.1145/2838344.2852078.

[13] Gavin. *Infiniband vs. RoCE V2: A comparison of network architectures for AI Computing Centers - NADDOD blog*. 2023. URL: https://www.naddod.com/blog/infiniband-vs-roce-v2-which-is-best-network-architecture-for-ai-computing-center.

[14] Juncheng Gu et al. "Efficient Memory Disaggregation with Infiniswap". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 649–667. ISBN: 978-1-931971-37-9. URL: `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu`.

[15] Jim Handy. *The memory/storage hierarchy*. 2020. URL: `https://thessdguy.com/the-memory-storage-hierarchy/`.

[16] Jim Handy. *Two CXL conundrums*. 2023. URL: `https://thememoryguy.com/two-cxl-conundrums/#:~:text=It%20has%20to%20do%20with,because%20it%20requires%20a%20controller`.

[17] *Hazelcast*. 2024. URL: `https://hazelcast.com/`.

[18] Torsten Hoefler. *What are the real differences between RDMA, Infiniband, RMA, and PGAS?* 2016. URL: `https://htor.inf.ethz.ch/blog/index.php/2016/05/15/what-are-the-real-differences-between-rdma-infiniband-rma-and-pgas/#:~:text=RDMA%20is%20more%20powerful%20than,do%20not%20include%20each%20other!`.

[19] Song Jiang, Feng Chen, and Xiaodong Zhang. "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement." In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 323–336.

[20] Song Jiang and Xiaodong Zhang. "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance". In: *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '02. Marina Del Rey, California: Association for Computing Machinery, 2002, 31–42. ISBN: 1581135319. DOI: `10.1145/511334.511340`. URL: `https://doi.org/10.1145/511334.511340`.

[21] Theodore Johnson, Dennis Shasha, et al. "2Q: a low overhead high performance bu er management replacement algorithm". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. Citeseer. 1994, pp. 439–450.

[22] Anuj Kalia, Michael Kaminsky, and David G Andersen. "Using RDMA efficiently for key-value services". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. 2014, pp. 295–306.

[23] Donghee Lee et al. "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies". In: *IEEE transactions on Computers* 50.12 (2001), pp. 1352–1361.

[24] Huaicheng Li et al. *Pond: CXL-Based Memory Pooling Systems for Cloud Platforms*. 2022. arXiv: `2203.00241 [cs.OS]`.

[25] YingQi Li, Meiju Yu, and Ru Li. "A Cache Replacement Strategy Based on Hierarchical Popularity in NDN". In: *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*. 2018, pp. 159–161. DOI: `10.1109/ICUFN.2018.8436597`.

[26] Hyeontaek Lim et al. "{MICA}: A holistic approach to fast {In-Memory}{Key-Value} storage". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 429–444.

[27] Chengzhi Lu et al. "Imbalance in the cloud: An analysis on Alibaba cluster trace". In: *2017 IEEE International Conference on Big Data (Big Data)*. 2017, pp. 2884–2892. DOI: `10.1109/BigData.2017.8258257`.

[28] Michael Marty et al. "Snap: a Microkernel Approach to Host Networking". In: *In ACM SIGOPS 27th Symposium on Operating Systems Principles*. New York, NY, USA, 2019.

[29] Hasan Al Maruf and Mosharaf Chowdhury. "Memory Disaggregation: Advances and Open Challenges". In: (2023). arXiv: `2305.03943 [cs.DC]`.

[30] Dhruv Matani, Ketan Shah, and Anirban Mitra. *An O(1) algorithm for implementing the LFU cache eviction scheme*. 2021. arXiv: `2110.11602 [cs.DS]`.

[31] Nimrod Megiddo and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache". In: *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. San Francisco, CA: USENIX Association, Mar. 2003. URL: `https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache`.

[32] *Memcached*. 2024. URL: `https://memcached.org/`.

[33] Jai Menon. *Council post: The rise of memory-centric architectures*. 2018. URL: `https://www.forbes.com/sites/forbestechcouncil/2018/11/16/the-rise-of-memory-centric-architectures/?sh=3ed59d6c5952`.

[34] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using {One-Sided}{RDMA} Reads to Build a Fast,{CPU-Efficient}{Key-Value} Store". In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013, pp. 103–114.

[35] Kathlene Morales and Byeong Kil Lee. "Fixed Segmented LRU cache replacement scheme with selective caching". In: *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*. 2012, pp. 199–200. DOI: `10.1109/PCCC.2012.6407712`.

[36] Timothy Prickett Morgan. *CXL And Gen-Z Iron Out A Coherent Interconnect Strategy — nextplatform.com*. `https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/`. 2020.

[37] Rajesh Nishtala et al. "Scaling memcache at facebook". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 385–398.

[38] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. "The LRU-K page replacement algorithm for database disk buffering". In: *Acm Sigmod Record* 22.2 (1993), pp. 297–306.

[39] *Redis*. 2024. URL: `https://redis.io/`.

[40] Audrey Rawnie Rico. *40+ essential google search statistics for 2024*. 2024. URL: `https://fitsmallbusiness.com/google-search-statistics/#:~:text=There%20are%208.5%20billion%20Google%20searches%20per%20day&text=Google%20runs%20around%2099%2C000,or%20four%20times%20per%20day.`.

[41] Gary Ruggles, Madhumita Sanyal, and Richard Solomon. *What is Compute Express Link (CXL) 3.0? – data center memory: Synopsys blog*. 2022. URL: `https://www.synopsys.com/blogs/chip-design/what-is-compute-express-link-3.html`.

[42] Ricardo Santana et al. "To ARC or Not to ARC". In: *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*. Santa Clara, CA: USENIX Association, July 2015. URL: `https://www.usenix.org/conference/hotstorage15/workshop-program/presentation/santana`.

[43] Jiacheng Shen et al. "Ditto: An elastic and adaptive memory-disaggregated caching system". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 675–691.

[44] Jiacheng Shen et al. "Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. <conf-loc>, <city>Koblenz</city>, <country>Germany</country>, </conf-loc>: Association for Computing Machinery, 2023, 675–691. ISBN: 9798400702297. DOI: `10.1145/3600006.3613144`. URL: `https://doi.org/10.1145/3600006.3613144`.

[45] Arjun Singhvi et al. "1rma: Re-envisioning remote memory access for multi-tenant datacenters". In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 708–721.

[46] Ruihong Wang et al. "The case for distributed shared-memory databases with rdma-enabled memory disaggregation". In: *arXiv preprint arXiv:2207.03027* (2022).

[47] Zhonghua Wang et al. "Rcmp: Reconstructing RDMA-Based Memory Disaggregation via CXL". In: *ACM Trans. Archit. Code Optim.* 21.1 (2024). ISSN: 1544-3566. DOI: `10.1145/3634916`. URL: `https://doi.org/10.1145/3634916`.

[48] Hugh E. Williams. *Caching in search*. 2012. URL: `https://hughewilliams.com/2012/05/24/caching-in-search/`.

[49] Juncheng Yang. *FIFO queues are all you need for cache eviction*. 2023. URL: `https://blog.jasony.me/system/cache/2023/08/01/s3fifo`.

[50] Juncheng Yang, Yao Yue, and K. V. Rashmi. "A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter". In: *ACM Trans. Storage* 17.3 (2021). ISSN: 1553-3077. DOI: `10.1145/3468521`. URL: `https://doi.org/10.1145/3468521`.

[51] Juncheng Yang et al. "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*. HOTOS '23. Providence, RI, USA: Association for Computing Machinery, 2023, 70–79. ISBN: 9798400701955. DOI: `10.1145/3593856.3595887`. URL: `https://doi.org/10.1145/3593856.3595887`.

[52] Juncheng Yang et al. "FIFO queues are all you need for cache eviction". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. <conf-loc>, <city>Koblenz</city>, <country>Germany</country>, </conf-loc>: Association for Computing Machinery, 2023, 130–149. ISBN: 9798400702297. DOI: `10.1145/3600006.3613147`. URL: `https://doi.org/10.1145/3600006.3613147`.

[53] Farzana Beente Yusuf et al. "Cache Replacement as a MAB with Delayed Feedback and Decaying Costs". In: *arXiv preprint arXiv:2009.11330* (2020).

[54] Yazhuo Zhang et al. "Sieve is simpler than lru: an efficient turn-key eviction algorithm for web caches". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). USENIX Association*. 2024.