Designing Cloud Caches over Hierarchical Storage

Lorenzo Marini, Damiano Carra
Computer Science Department
University of Verona, Italy
{name.surname}@univr.it

***Abstract*—Cloud caches are essential for reducing latency in modern web services and are typically implemented with DRAM-based Key-Value stores such as Redis or Memcached. Although elastic scaling allows these caches to adapt to changing demand, current optimization strategies usually focus on a single tier, disregarding different combinations of storage types.**

In this paper, we introduce a cost-effective hierarchical cloud caching architecture that minimizes the total operating cost of a cloud cache for a target serving delay. Our design combines an expensive DRAM-based cache with a high-capacity, low-cost object storage layer, coupled with a dynamic cache management scheme that jointly optimizes storage allocation and admission decisions. By using the lower tier to buffer warm objects and to host large objects, the system reduces origin traffic, limits unnecessary DRAM admissions, and reserves the fast tier for latency-critical content. Experiments with real-world traces show that the proposed hierarchy can significantly reduce total operating cost compared with traditional single-tier cloud caching solutions while maintaining competitive latency.

I. Introduction

Cloud caches are a fundamental building block of many services [1] . They provide fast access to popular content, reduce the delay perceived by users, and are commonly implemented with DRAM-based Key-Value stores such as Redis [2] or Memcached [3], which can sustain very low access latencies.

Cloud platforms further make these solutions attractive because they support dynamic scaling [4]. In practice, the size of the cache can be increased or decreased according to the predicted load, with the goal of meeting a target performance metric such as hit ratio or average serving delay while keeping the deployment cost under control. This flexibility is valuable, but it also introduces an important trade-off: reducing the cache size lowers storage cost, yet the resulting increase in misses directly affects the delay experienced by requests.

This trade-off is particularly harmful when the cache adopts management policies such as LRU, which admit an object on every miss. If requests for an object are too infrequent, the object may be admitted, evicted, and later admitted again, generating repeated misses and wasting expensive DRAM capacity. In addition, many workloads contain objects that are requested only once, usually referred to as *one hit wonders* [5]. These objects consume precious cache space without providing reuse. Policies that limit their impact, such as 2-LRU [6], delay admission until the second request, but they also introduce a different cost: objects that are eventually reused must still be fetched twice before they can benefit from the cache.

A second difficulty arises from the highly heterogeneous sizes of the objects served by web caches [5]. Large objects can occupy a disproportionate fraction of the DRAM-based cache and may force the eviction of many smaller objects that collectively contribute more to the hit ratio. As a result, a single-tier DRAM cache must simultaneously deal with one-hit wonders, reusable warm objects, and large files, even though these classes have very different implications for cost and latency.

In this paper, we consider a hierarchical cloud storage design to reduce the overall cost of cache management. In addition to storage charges, some cloud storage services also incur per-operation costs, typically for `get()` and `set()` operations. While storage hierarchies have been studied extensively in other contexts [7], they have not been systematically explored in the context of *cloud caches*. We pair an expensive DRAM-based cache with a cheaper HDD-based storage layer, *e.g.* object storage, that keeps warm and cold objects. In a policy such as 2-LRU, an object can be placed in the HDD tier upon the first request and promoted to DRAM on the second request. This reduces latency relative to fetching the object again from the origin, limits the impact of one-hit wonders on the DRAM tier. The presence of an additional storage level also helps with the management of large objects. Web caches often serve objects whose sizes vary by orders of magnitude. Keeping the largest objects in the HDD tier preserves DRAM capacity for the objects that benefit most from very low latency, with only a limited impact on end-to-end delay.

The dynamic scheme we propose jointly considers DRAM and HDD storage cost, HDD operation cost, predicted demand, object sizes, and cache admission and eviction policies to provide a cost-effective cloud caching solution. Our contribution is therefore twofold: we define a hierarchical cache-management framework tailored to cloud pricing models, and we show through trace-driven evaluation that this framework can significantly reduce total cost compared with current single-tier designs while still meeting latency targets.

II. Background and Related Work

Cloud caches. Cloud computing enables services to be instantiated and resized dynamically as traffic fluctuates. In web

architectures, for example, the number of front-end servers can be adjusted to accommodate changes in demand, and the same elasticity is available for the caching layer. Among the managed services offered by cloud providers, in-memory data stores used as caches play a central role. Notable examples include Amazon's ElastiCache [8], Microsoft's Azure Redis Cache [9], and Google's Cloud Memorystore [10].

These managed services handle most cache-related operations and expose simple APIs to create and terminate instances and to manage clusters. Users can choose among multiple configurations for each instance, and both the instance type and the provisioned capacity contribute to the overall cost of operating a cloud cache. In practice, however, the price of the fast tier is only part of the picture: the way misses are handled, and the type of storage used to absorb them, can materially change both latency and operating cost.

TABLE I: Representative pricing for three public cloud providers (European central region) for HDD-based storage (object storage) and DRAM-based storage. These options are typically charged for by the hour.

Operation	**AWS**	**Azure**	**GCP**
HDD storage (per GB/month)	2.3¢	2.1¢	2.3¢
DRAM storage (per GB/month)	700-1200 ¢		
Object `get()` (per 1k reqs)	0.04¢	0.05¢	0.04¢
Object `set()` (per 1k reqs)	0.5¢	0.65¢	0.5¢

Storage hierarchy. Cloud providers offer several storage options, each with different performance characteristics. In principle, memory and storage devices can be organized into a *hierarchy* of tiers, with faster devices above slower and often larger ones. In practice, however, the gap between emerging technologies such as SSDs and NVMe devices is not always sharp, and their latency and bandwidth characteristics often overlap. The authors in [7] survey these alternatives and discuss the trade-offs they imply.

In our setting, as described in Sec. III, we focus on a two-level hierarchy composed of a fast, expensive DRAM tier and a slower, cheaper HDD-based tier. This choice is motivated not only by the difference in access time, but also by the large difference in cost shown in Table I. Besides capacity cost, the lower tier also incurs per-operation cost, mainly for `get()` and `set()`, which must be included in the evaluation. A hierarchical design is therefore useful only if it reduces origin traffic and DRAM occupancy enough to compensate for these additional costs.

Miss Ratio Curves and Cache policies A Miss Ratio Curve (MRC) [11] represents, for a given request stream, the miss ratio (and, consequently, the hit ratio) of a cache as a function of its size. Early work on MRC computation assumed uniform object sizes and a limited set of eviction policies (*e.g.*, LRU, LFU). More recent approximate methods rely on sampling [12], [13], [14] or approximate computation [15]. MRCs can also be constructed for workloads with heterogeneous object sizes [16] and for different eviction policies—in the latter case, an alternative approach based on Miniature Simulations [14], [17] is required.

The ability to build MRCs for different policies is important because no single policy performs best for every traffic pattern. System administrators may therefore want to evaluate multiple policies for the workloads they serve before deciding how much storage to provision. A caching policy is defined by both its *admission* and *eviction* rules, *i.e.*, the criteria used to accept an object into the cache and to remove one when space is needed. While LRU has no explicit admission rule—every miss leads to admission—policies such as 2-LRU [6] use a shadow cache and admit an object only after it has been requested at least twice. In our setting, explicitly separating admission and eviction is especially useful because each decision has a direct impact on storage consumption, operation cost, and delay.

Related Work. Elastic cloud caching has been studied extensively in the literature; see, for example, [18], [4], [19], [20], [21]. These works focus on the single-tier case, where a DRAM-based cache is dynamically resized to meet a target performance objective, and they typically evaluate only a limited set of cache management policies.

Other examples that aim to minimize cloud cost include [22] and [23], where the authors explore the use of spot instances for different goals, such as content replication or lower storage cost. These solutions also consider essentially a single tier. The authors in [24] explore multiple storage options, but their work is not focused on storage used as caches and does not analyze how cache management policies affect serving delay.

Storage hierarchies have long been studied in computer systems—see [7] and the references therein. Most of this literature, however, does not target a cloud setting with explicit storage and operation charges. The authors in [25] study the impact of hierarchies on the cost of managing a cross-region storage service with the help of a cache, but they do not explicitly study a cloud caching service whose primary goal is to serve requests at low latency. Cloud caches sometimes offer elastic services with the help of secondary storage, but they do not offer any means for controlling the average delay or the costs in detail. These are the gaps addressed by our work.

III. Hierarchical Cache Management

Figure 1 provides a high-level view of the proposed solution and of the main decisions that must be taken to design it. The key design knobs concern the number of hierarchy levels, the objects that should be stored in each level, and the rules used to move objects across levels while the overall available capacity changes over time.

In the following, we discuss the main design alternatives and motivate the choices we settled on. The guiding principle throughout this section is that the lower tier should not simply act as a slower copy of the fast cache; rather, it should absorb the objects that are too costly to keep in DRAM all the time while still contributing enough reuse to justify their presence in the hierarchy.

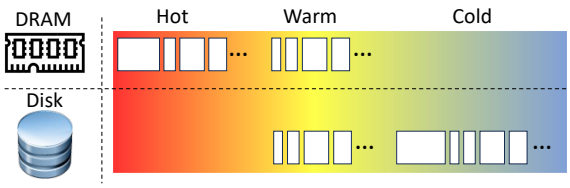


Fig. 1: General scheme of our solution.

Number of levels. There are many storage technologies available in the cloud, each characterized by its access delay, transfer bandwidth, and cost. In principle, one could rank these options and build a hierarchy with many levels. Such a design would provide additional flexibility, but it would also substantially increase management complexity, because the system would need to decide where each object should be stored, whether the object should be replicated, and how the capacity of each level should evolve over time.

This complexity becomes even more pronounced when dynamic resizing is considered. A multi-level design would require a resizing rule for every tier, along with migration rules across adjacent tiers, making both the optimization problem and the implementation harder. Moreover, the latency seen by requests depends not only on the raw access time of each storage technology, but also on the processing overhead introduced by traversing the hierarchy and by executing the policy logic at each level.

The ultimate goal of the caching service is to provide content with a target latency while maintaining low operating cost. After exploring different alternatives, we found that a simple design with two levels offers the best trade-off between expressiveness and practicality. In particular, we use the two ends of the cloud-storage spectrum that matter most for our objective: a DRAM-based tier for low-latency service, and a low-cost HDD-based tier for storing objects that may still be useful but do not justify permanent residence in DRAM.

Tiering vs mirroring. The presence of multiple levels requires a policy to decide *where* objects should be stored. In caching, management policies such as LRU or LFU identify the objects that are more likely to be requested again and those that can be safely evicted if space becomes insufficient. Their ranking can therefore be used to place objects across levels.

If we impose that each object can be stored in exactly one level, we adopt an approach usually referred to as *tiering*. Strict tiering uses storage efficiently, but it introduces an additional delay whenever an object must be moved from one level to another before being served. At the opposite extreme, one can mirror the same content across multiple levels, but this increases storage usage and therefore cost.

Our approach is inspired by [7] and identifies three coarse-grained classes of objects, labeled as "hot", "warm", and "cold". Differently from previous approaches [7], [5], however, we mirror only the warm objects. The rationale is that warm objects have an intrinsically ambiguous role: they may correspond either to content that is still requested often enough to deserve fast access, or to content whose popularity is decreasing and that will soon become cold. Mirroring them reduces the delay of subsequent accesses, smooths the transition between tiers, and makes resizing less disruptive, while avoiding the full cost of replicating the entire lower tier.

Dynamic resizing. The size of the cache depends on the characteristics of the request pattern. The relation between cache size and hit ratio is not linear and exhibits diminishing returns: increasing the cache indefinitely does not imply an equally large improvement in hit ratio. This relation is commonly captured by the Miss Ratio Curve (MRC)—see Sec. II.

In a dynamic context, where the request pattern changes over time, the standard approach is to observe traffic over an interval, build the MRC, and then select the cache size for the next interval. MRCs can be built efficiently for a broad class of cache policies, with uniform or non-uniform object sizes, using tools proposed in the literature [17]. This makes online resizing practical even when the cache policy is more sophisticated than plain LRU.

In our framework, since we are dealing with multiple tiers, we can adopt schemes that can build a multi-dimensional MRC as described in [26]. We select the combination of DRAM-based and HDD-based storage that attains the target average delay[1] at the minimum cost. The expected delay is derived from the measured delays of serving requests from DRAM, from HDD, and from the origin, together with the hit ratios predicted for the next interval. The optimization therefore balances storage cost, the impact of misses on delay, and the additional operation cost incurred by the lower tier.

Cache management policy. The central element that defines a cache is its management policy, which is composed of both admission and eviction decisions. In a setting with hierarchical storage and dynamic resizing, we must also specify how that policy is adapted across tiers. For example, the admission rule at the higher level may differ from the admission rule at the lower level, because admitting an object to the lower tier can be coupled with its eviction from the upper tier.

As noted above, no policy proposed in the literature works well in every possible scenario and under every traffic pattern. For this reason, in this section we design a policy for hierarchical storage that targets a traffic pattern commonly found in web-based systems: object popularities are Zipf-like, the popularity distribution has a long tail, and object popularity evolves over time, *i.e.*, requests for a given object are often concentrated within a limited time span.

In this context, LRU is the natural reference policy. Nevertheless, it has two important limitations. First, many objects are requested only once—the *one-hit wonders* discussed earlier. These objects pollute the cache and provide no benefit. A standard way to mitigate the problem is to use 2-LRU, which admits an object only on the second request. This can be implemented by maintaining a compact data structure that

[1]While we refer to the average delay as a possible target, other performance indices, such as the 99th percentile, could also be considered.

records the identifiers of objects seen once and updates that structure when an object is requested again.

Second, LRU is known not to provide the best hit ratio when objects have heterogeneous sizes. A classical solution is to adopt policies such as Greedy Dual Size (GDS) [27], which preserve the qualitative behavior of LRU while taking object size into account. GDS, however, still admits objects on the first request, so one-hit wonders can still consume expensive capacity. Despite the large literature on variants of GDS, to the best of our knowledge, none is specifically designed to address one-hit wonders within a hierarchical cloud cache.

For these reasons, we adopt a policy inspired by 2-LRU but enriched with size-awareness in the spirit of GDS, and we call it 2-GDS. The policy is designed explicitly for a two-level hierarchy, and it exploits the fact that the HDD tier is cheap enough to store potential one-hit wonders and other objects whose future reuse is still uncertain. This makes the lower tier a natural place for staging objects before they earn a place in DRAM.

The key points of the policy are summarized as follows:

- The first time an object is requested, we retrieve it from the source (since it needs to be served) and we store it in the cloud HDD.
- The second time an object is requested, if it is still in the HDD, we retrieve it from the HDD, and we put it in the DRAM. If it has been evicted from the HDD, we treat it as if it had been requested for the first time.
- The objects in the DRAM are managed with GDS, which determines the order that will be used in case of eviction from the DRAM.
- The last portion of the ordered GDS queue in the DRAM (warm objects) are replicated in the HDD, along with the objects that have been previously evicted (cold objects), based on the available space on the HDD;
- Very large objects, whose size is above a defined threshold, are always kept in the HDD.

The last point on large objects follows from practical considerations about the delay experienced by users. Admitting a very large object into DRAM may trigger the eviction of many small objects, thereby increasing the average delay. Conversely, when a large object is served from HDD, the initial access delay is amortized by the serving time. We therefore use a simple threshold to determine which objects are too large for DRAM. Using the HDD delay model in [28] and requiring the transfer delay at 100 Mbps to be at least 10 times the initial HDD seek time, objects with size larger than 1.0 MByte can be stored in HDD with a limited impact on the perceived delay[2].

Algorithm 1 provides the pseudocode of the proposed scheme (the management of large objects is not shown for simplicity). The design uses three logical regions of the cache, dedicated to hot, warm, and cold objects ($\mathcal{H}$, $\mathcal{W}$, and $\mathcal{C}$ respectively). The values of $\mathcal{H}+\mathcal{W}$ and $\mathcal{W}+\mathcal{C}$ are derived from the MRC analysis and from the costs associated with DRAM and HDD storage, respectively. The only degree of freedom is the size reserved for warm objects. In our experiments we observed that such size does not have a large impact as long as it remains below 30% of the DRAM space ($\mathcal{H} + \mathcal{W}$).

[2]An alternative approach could be to split the object into two portions: one small to be cached in the DRAM, and the rest in the HDD. In this way, the first part is served quickly and the user experience is improved.

Algorithm 1: Cache management scheme

input: $\mathcal{H}$, ordered dict. with "hot" objects, size S_H
input: $\mathcal{W}$, ordered dict. with "warm" objects, size S_W
input: $\mathcal{C}$, ordered dict. with "cold" objects, size S_C

$i \leftarrow$ PROCESSREQUEST(r_t) ;
if $i \notin \mathcal{H}, \mathcal{W}, \mathcal{C}$ **then**
  $o \leftarrow$ RETRIEVEOBJECT(i);
  $\mathcal{C}$.set(i, o);
  while $S_C^{curr} > S_C$ **do**
    $\mathcal{C}$.evict();
  return(o);
if $i \in \mathcal{H}$ **then**
  $o \leftarrow \mathcal{H}$.remove(i);
if $i \in \mathcal{W}$ **then**
  $o \leftarrow \mathcal{W}$.remove(i);
if $i \in \mathcal{C}$ **then**
  $o \leftarrow \mathcal{C}$.remove(i);
$\mathcal{H}$.set(i, o);
while $S_H^{curr} > S_H$ **do**
  $(e_k, e_v) \leftarrow \mathcal{H}$.evict();
  $\mathcal{W}$.set(e_k, e_v);
while $S_W^{curr} > S_W$ **do**
  $(e_k, e_v) \leftarrow \mathcal{W}$.evict();
  $\mathcal{C}$.set(e_k, e_v);
while $S_C^{curr} > S_C$ **do**
  $(e_k, e_v) \leftarrow \mathcal{C}$.evict();
// Return the object
return(o);

The solution proposed in Algorithm 1 does not assume a specific admission or eviction policy. Instead, it relies on standard APIs for admitting an object (`set()`), deleting an object (`remove()`), and evicting an object (`evict()`). It is also possible to use different policies for the two storage types, DRAM and HDD. In our experiments, we use GDS, but other policies, such as LRU or SIEVE [29], can be adopted easily. The scheme does encode the choice that the first time an object is requested it is stored in the HDD, but this behavior can be modified straightforwardly by storing the object directly in DRAM on its first request. More generally, the proposed framework can be extended or adapted to different traffic patterns and can serve as a reference design for hierarchical cloud caches.

Overall scheme. In summary, the proposed scheme periodically builds the MRC and selects the sizes of the DRAM and HDD tiers that meet a target average serving delay at minimum cost. When the size of the DRAM tier is reduced, objects are evicted from DRAM and stored in the HDD, which may in turn trigger additional evictions from the HDD. Thanks to the replication of warm objects, this resizing process can

be carried out without disrupting normal cache operation. If the HDD size is reduced, the system simply continues evicting objects until the target size is reached. The same framework can therefore support online adaptation without changing the core cache logic.

IV. Experimental Results

Methodology and settings. We evaluate our solution with a trace-driven approach, where we use a set of publicly available traces and emulate the behavior of the caching system under the request streams contained in those traces. The traces include requests for objects with non-uniform sizes. The first, `cdn`, comprises requests collected from a Content Delivery Network (CDN) [30], [31] serving photos and other media content for Wikipedia pages. The second, labeled as `systor` [32], is a block I/O trace from the SNIA IOTTA repository [33]. The third, `twitter` [34], [35], captures accesses from in-memory cache clusters and was collected in 2020.

Our primary performance metric is the average cost of operating the cloud cache for a target average delay. The average delay depends on the hit ratios of the DRAM-based and HDD-based caches: content served by the fast tier typically incurs only a few milliseconds, while content served by the HDD tier incurs a few tens of milliseconds (for example, around 20 ms), and content retrieved from the origin may incur 50–60 ms. The cost includes the storage cost of DRAM and HDD and the operation cost paid on the HDD tier (see Table I). In this evaluation, egress costs are the same for every solution because each request must eventually be served regardless of where the content is retrieved.

As a baseline, we consider a purely DRAM-based cloud cache solution. For both the baseline and our solution, we evaluate different cache management policies (SIEVE, 2-LRU, GDS, and 2-GDS), which generate different hit ratios and therefore different costs. To the best of our knowledge, the literature does not provide a hierarchical cloud caching baseline with the same objective and degree of flexibility, so a comparison with an existing multi-tier system is not possible.

Results. Figure 2 shows, for different target average delays, the average weekly cost of our solution compared with the DRAM-based baseline (averages over multiple weeks). Our scheme selects the combination of DRAM-based and HDD-based capacity needed to meet a specific delay target. As expected, tighter delay targets require more DRAM, because more requests must be served by the fastest tier. Even in that regime, however, the hierarchical solution still provides significant cost savings, since the lower tier absorbs warm objects and large objects that would otherwise occupy expensive DRAM capacity. We observe the same qualitative trend for the `twitter` trace and for different delay distributions, which are omitted here for space constraints.

As noted earlier, the average delay is ultimately determined by the hit ratios of the different tiers. Figure 3 shows representative Miss Ratio Curves (MRCs) for the `cdn` and `systor` traces under different cache management policies.

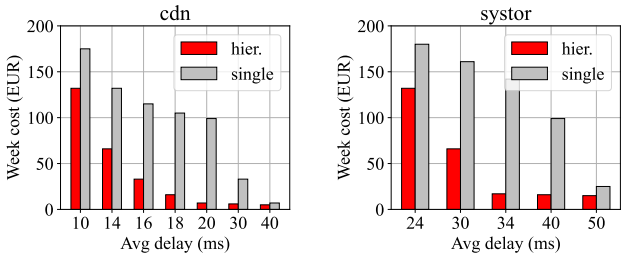


Fig. 2: Average weekly costs for different target delays (cache management policy used is 2-GDS).

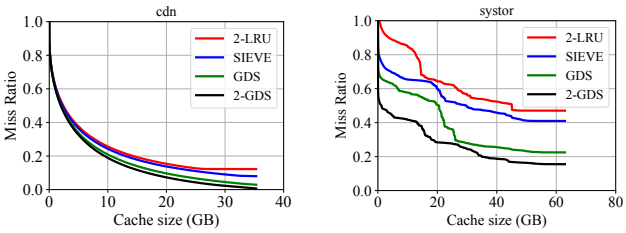


Fig. 3: Representative Miss Ratio Curves for two traces, for different cache management policies.

These curves help explain why the cost of the system varies not only with the amount of provisioned storage, but also with the admission and eviction rules used to populate the hierarchy.

The `cdn` trace is more regular, and therefore the different policies yield rather similar MRCs, although noticeable gains in hit ratio can still be obtained at small cache sizes. By contrast, the `systor` trace shows larger differences among the MRCs, which translate into more significant gains or losses in hit ratio. Overall, the choice of cache management policy remains important, because every improvement in hit ratio directly affects both the achievable delay and the operating cost of the cache.

V. Discussion and Conclusion

Cloud caches represent a building block for providing web-based services with low latency. The dominant approach today relies on a single fast tier that serves the hottest content. Exploring a broader set of storage options offered by cloud providers opens a non-trivial design space, but it also creates an opportunity to exploit cloud pricing more effectively.

In this paper, we proposed a multi-tier solution designed to work with heterogeneous object sizes and different cache management policies. The key idea is to use the lower tier not merely as overflow space, but as an active component of the cache that buffers warm objects, filters one-hit wonders, and hosts large objects. Our results show that this design can significantly reduce cost for a given target average delay compared with a single-tier approach. A possible direction for future work is to study a network of caches, where objects are distributed across multiple cloud cache instances.

REFERENCES

[1] D. Chiu, A. Shetty, and G. Agrawal, "Elastic cloud caches for accelerating service-oriented computations," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[2] Redis, "Redis," https://redis.io/, accessed: Apr. 2026.

[3] Memcached, "Memcached," https://memcached.org/, accessed: Apr. 2026.

[4] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi, "Elastic caching." in *SODA*, 2019, pp. 143–156.

[5] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "{AdaptSize}: Orchestrating the hot object memory cache in a content delivery network," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 483–498.

[6] M. Garetto, E. Leonardi, and V. Martina, "A unified approach to the performance analysis of caching systems," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 1, no. 3, pp. 1–28, 2016.

[7] K. Tu, K. Wu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Getting the {MOST} out of your storage hierarchy with {Mirror-Optimized} storage tiering," in *24th USENIX Conference on File and Storage Technologies (FAST 26)*, 2026, pp. 561–578.

[8] AWS, "Amazon Web Service ElastiCache," https://aws.amazon.com/elasticache/, accessed: Apr. 2026.

[9] Microsoft, "Azure Cache for Redis," https://azure.microsoft.com/en-us/products/cache/, accessed: Apr. 2026.

[10] Google, "Cloud Memorystore," https://cloud.google.com/memorystore/, accessed: Apr. 2026.

[11] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970.

[12] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 351–364.

[13] C. A. Waldspurger, N. Park, A. T. Garthwaite, and I. Ahmad, "Efficient MRC Construction with SHARDS," in *FAST*, 2015, pp. 95–110.

[14] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *Proceedings of USENIX ATC*, 2017, pp. 487–498.

[15] J. R. Murillo, G. Totoy, and C. L. Abad, "Instrumenting cloud caches for online workload monitoring: The case of online miss rate curve estimation in memcached," in *Proceedings of the 16th Workshop on Adaptive and Reflective Middleware*, 2017, pp. 1–6.

[16] D. Carra and G. Neglia, "Efficient miss ratio curve computation for heterogeneous content popularity," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 741–751.

[17] K. Shakiba, S. Sultan, and M. Stumm, "Kosmo: Efficient online miss ratio curve generation for eviction policy evaluation," in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, 2024, pp. 89–105.

[18] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching." in *HotStorage*, 2015.

[19] X. Andrade, J. Cedeno, E. Boza, H. Aragon, C. Abad, and J. Murillo, "Optimizing cloud caches for free: A case for autonomic systems with a serverless computing approach," in *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2019, pp. 140–145.

[20] D. Carra, G. Neglia, and P. Michiardi, "Elastic provisioning of cloud caches: A cost-aware ttl approach," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1283–1296, 2020.

[21] J. Zhang, A. Wang, X. Ma, B. Carver, N. J. Newman, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan *et al.*, "Infinistore: Elastic serverless cloud storage," *arXiv preprint arXiv:2209.01496*, 2022.

[22] Z. Xu, C. Stewart, N. Deng, and X. Wang, "Blending on-demand and spot instances to lower costs for in-memory storage," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.

[23] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang, "Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 620–634.

[24] H. Park, G. R. Ganger, and G. Amvrosiadis, "Mimir: Finding cost-efficient storage configurations in the public cloud," in *Proceedings of the 16th ACM International Conference on Systems and Storage*, 2023, pp. 22–34.

[25] H. Park, Z. Qiu, G. R. Ganger, and G. Amvrosiadis, "Reducing cross-cloud/region costs with the auto-configuring macaron cache," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 347–368.

[26] Z. Liu, H. W. Lee, Y. Xiang, D. Grunwald, and S. Ha, "{eMRC}: Efficient miss ratio approximation for {Multi-Tier} caching," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 293–306.

[27] P. Cao and S. Irani, "{Cost-Aware}{WWW} proxy caching algorithms," in *USENIX Symposium on Internet Technologies and Systems (USITS 97)*, 1997.

[28] G. Neglia, D. Carra, M. Feng, V. Janardhan, P. Michiardi, and D. Tsigkari, "Access-time-aware cache algorithms," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 2, no. 4, pp. 1–29, 2017.

[29] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. Rashmi, "{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)*, 2024, pp. 1229–1246.

[30] Z. Song, D. S. Berger, K. Li, A. Shaikh, W. Lloyd, S. Ghorbani, C. Kim, A. Akella, A. Krishnamurthy, E. Witchel *et al.*, "Learning relaxed belady for content distribution network caching," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 529–544.

[31] Z. S. et at. (2020) Cdn traces. [Online]. Available: https://github.com/sunnyszy/lrb

[32] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. ACM, 2017, pp. 13:1–13:11.

[33] SNIA. (2008) Snia iotta repository block i/o traces. [Online]. Available: http://iotta.snia.org/traces/block-io

[34] J. Yang, Y. Yue, and K. Rashmi, "A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter," *ACM Transactions on Storage (TOS)*, vol. 17, no. 3, pp. 1–35, 2021.

[35] J. Y. et at. (2020) Twitter production cache traces. [Online]. Available: https://github.com/twitter/cache-trace