Investigating Multi-tier and QoS-aware Caching based on ARC

Lydia Ait-Oucheggou, Stéphane Rubini, Abdella Battou, Jalil Boukhobza

Investigating Multi-tier and QoS-aware Caching based on ARC

Lydia Ait-Oucheggou*‡, Stéphane Rubini†, Abdella Battou*, Jalil Boukhobza‡
*NIST, Maryland, USA,
†Univ. Bretagne Occidentale, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
‡ENSTA Bretagne, Lab-STICC, CNRS, UMR 6285, F-29200 Brest. France
Email: lydia.ait_oucheggou@ensta-bretagne.org, stephane.rubini@univ-brest.fr,
abdella.battou@nist.gov, jalil.boukhobza@ensta-bretagne.fr

***Abstract*—Memory caching is a common practice to reduce application latencies by buffering relevant data in high speed memory. When the volume of data to cache is too large or a DRAM-based solution too expensive, several technologies such as NVM or high speed SSDs could complement DRAM to form a multi-tier cache. Additionally, most existing policies focus on categorizing the data based on factors like recency and frequency, setting aside the fact that applications/customers have varying Quality-of-Service requirements. This concept is well established in Cloud environment with Service Level Agreement (SLA). In this paper, by extending the Adaptive Replacement Cache (ARC), that uses recency and frequency lists, we propose a QoS-aware Multi-tier Adaptive Replacement Cache (QM-ARC) policy with the ability to take into account data applications/customers priorities through the concept of penalty borrowed from the Cloud. QM-ARC is generic, as it can be applied whatever the number of tiers and can accommodate different penalty functions. Using synthetic and real traces, our solution improved QoS as compared to state-of-the-art work.**

***Index Terms*—Cache, Storage, ARC, Multi-tier, QoS**

I. Introduction

The demand for high-performance caching in applications like video streaming has resulted in an urging need for high-speed memory. However, as Internet data traffic continues to grow, these applications require larger, costly caches. On the other hand, the new breakthrough in storage and memory systems (e.g. Intel Optane) makes it possible to enlarge caches in cost-effective and energy-efficient ways [1]–[3].

Additionally, users express different levels of service expectations from their suppliers [4]. This is a well established practice in the Cloud with the concept of Service Level Agreement (SLA). Most state-of-the-art caching techniques are performance-driven and agnostic to variable QoS needs. Most previous studies focused on differentiating data based on factors such as popularity (LFU), recency (LRU), without considering QoS requirements per user or per application [5].

One of the most popular algorithms on caching policies is Adaptive Replacement Cache (ARC) [6]. ARC maintains two LRU lists: a recency list and a frequency list. The former contains objects that were requested only once, whereas the latter contains objects that were requested at least twice. ARC dynamically adapts the size of each list according to workload change. Even if several weaknesses of ARC have been pinpointed in state-of-the-art work [7], [8], it is still considered as a reference strategy that several recent propositions try to enhance. However, ARC does not take into consideration QoS.

Several state-of-the-art studies explored optimizations for ARC [9], [10]. However, it was focused on a single-tier configuration that did not take into consideration the QoS. Some interesting studies proposed novel so-called multi-tier caching strategies [11], [12]. However, those did not necessarily consider heterogeneous storage devices. In addition, they did not consider application QoS. Finally, caching studies that tried to handle QoS, did not consider multi-tier caches, and most of them did not evaluate the ability to satisfy QoS while their QoS model lacks flexibility [4], [13].

In this paper, we propose QM-ARC, a **Q**oS-aware, **M**ulti-tier **ARC** policy that upgrades ARC regarding two aspects:

i. QoS-based cache: QM-ARC borrows the concept of SLA and penalty-based resource management from the Cloud to help maintain data in the cache according to priority levels.

ii. Multi-tier caches: QM-ARC adapts to several different cache tiers through size adjustments and index selection for the insertions and promotions.

QM-ARC has been evaluated against some reference strategies and regarding several metrics related to cost, hit rate in different tiers and for different levels of priority, and it showed promising results, a decrease of 23% for penalty and increase of 40% in cache hit rate for high priority data.

II. Background on ARC

ARC [6] manages two LRU lists, see Figure 1 (gray part only) a recency list T_1 and a frequency list T_2. The Most Recently Used (MRU) objects that were accessed only once are kept in the recency list T_1, while objects that were accessed multiple times are stored in the frequency list T_2. With a cache size of c, ARC dynamically adjusts the number of pages allocated to each list by updating the size p of the list T_1. It does so by tracking the usage patterns of data using two LRU shadow lists: B_1 and B_2, which contain references to data evicted from T_1 and T_2, respectively. Virtually, $T_1+T_2+B_1+B_2 = 2c$. As B_1 and B_2 are shadow lists, they contain only data references. The algorithm dictates four cases:

Case 1: If a cache hit occurs in T_1 or in T_2, data are promoted to the MRU position in T_2.

Case 2: If a cache hit occurs in B_1, it means that the data was recently evicted from the cache but is still considered valuable. To prevent evicting the data, the ARC algorithm promotes it to T_2, effectively extending its "lifespan" in the cache. Simultaneously, the size p of T_1 is increased by $\max\{1, |B_2|/|B_1|\}$ to capture and retain these valuable data.

Case 3: If a cache hit occurs in B_2, it means that the data was recently evicted from the cache but is still considered valuable. To prevent evicting the data, the ARC algorithm promotes it to T_2, effectively extending its "lifespan" in the cache. Simultaneously, the size of T_2 is increased by $\max\{1, |B_1|/|B_2|\}$ to capture and retain these valuable data.

Case 4: If a miss occurs in all lists, data are placed in T_1.

When T_1 is full ($T_1 = p$) data are evicted in LRU fashion and their reference moved to the B_1 list to make room for new data. When T_2 becomes full ($T_2 = c - p$), data are evicted in LRU fashion and their reference added to B_2. If B_1 or B_2 are full, references are evicted and no longer tracked by ARC.

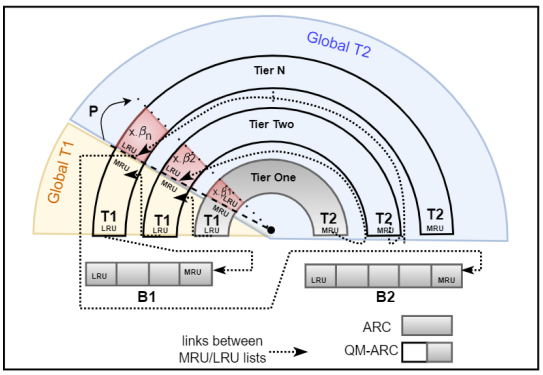


Fig. 1: ARC and QM-ARC global view

III. QM-ARC: QoS-aware Multi-tier ARC

A. Considering multi-tier

Our approach consists in applying ARC on a multi-tier cache. Each of the global lists T_1 and T_2 is sliced into a set of per-tier local lists, as illustrated in Figure 1. Each tier maintains its own local T_1 and T_2 lists, both organized in LRU fashion. The most efficient tier holds the MRU data.

The global list T_1 goes through continuous size p adjustments, and these adjustments are reflected proportionally in the tiers. This means that the size p_i of each local list T_i varies across tiers, but remains proportional to the size of the other lists in the different levels.

To achieve this proportional update, we introduce the term β_i, which represents the ratio of the size of the tier i to the size of the first (top) tier. Let p_i denote the size of the local list T_1 in tier i. When there is a cache hit in B_1, we increment the size of each p_i by β_i if the size of B_1 is greater than or equal to the size of B_2. If the size of B_1 is smaller than the size of B_2, we increment p_i by $(|B_2|/|B_1|) \cdot \beta_i$. However, all increments to p_i are bound by the maximum size of tier i. Similarly, when there is a cache hit in B_2, we decrement the size of each p_i by β_i if the size of B_2 is greater than or equal to the size of B_1. If the size of B_2 is smaller than the size of B_1, we decrement p_i by $(|B_1|/|B_2|) \cdot \beta_i$. However, all decrements to p_i are bounded by a minimum value of 0. Note that the ratios $(|B_1|/|B_2|)$ and $(|B_2|/|B_1|)$ come from the original ARC, see Section II.

The size p of the global list $T1$ is updated by β, which is the sum of all β_i values. Specifically, $\beta = \sum \beta_i = \sum (|tier_i|/|tier_1|)$. This ensures that the overall size adjustment of $T1$ is distributed proportionally among the tiers based on their relative sizes. By updating the sizes of the local lists in this manner, QM-ARC maintains a balanced allocation of cache resources across the tiers, adapting to the changing dynamics of the cache workload.

Regarding the two history lists, B_1 and B_2, they are shared across all tiers. However, in order to enhance the search efficiency within these lists, they are stored in the most efficient tier as an index towards indirect entries instead of being stored directly in each tier.

B. Considering QoS

Regarding the QoS management, QM-ARC assumes that data are assigned a priority level according to user categories. QM-ARC assumes that the data are tagged with a certain priority, and that a penalty function is associated with the data priorities. The penalty represents a cost to be paid by the service provider in case of late delivery of data. It can be a simple or complex function that fixes a penalty according to a given QoS metric. The penalty function used by QM-ARC assigns penalties based on the priority level of the data and the time it takes to retrieve them. Higher-priority data have higher penalties than low priority data when they undergo the same latency. We used the penalty concept from the Cloud and calculate it as in [14], [15] as follows (note that other functions could be used):

$$\textit{Penalty} = \sum_{k \in m} P_k$$

$$P_k(l) = \gamma k \times \begin{cases} 0 & \text{if } 0 \leq l < l_{SLA_{soft}} \\ P_{0_{\text{soft}}} & \text{if } l_{SLA_{soft}} \leq l < l_{SLA_{hard}} \\ P_{0_{\text{hard}}} & \text{if } l \geq l_{SLA_{hard}} \end{cases} \quad (1)$$

P_k represents the penalty function for the class of priority k, which is proportional by $\gamma_k \in]0, 1]$ to the penalty of the highest priority 0, i.e., $\gamma_k = P_{0_{hard}}/P_{k_{hard}}$. The higher the priority, the higher the γ, with $\gamma_k < \gamma_{k-1}$ and $\gamma_0 = 1$ represents the highest priority. The variable m is the total number of priority levels. l represents the latency it took to retrieve the data. l_{SLA} is the latency requested from the SLA per application/customer. As from Equation.1, when l is smaller than $l_{SLA_{soft}}$, no penalty is applied. When l is between $l_{SLA_{soft}}$ and $l_{SLA_{hard}}$, a medium penalty $\gamma_k \cdot P_{0_{soft}}$ is applied and when $l_{SLA_{hard}}$ is exceeded, a higher penalty $\gamma_k \cdot P_{0_{hard}}$ is applied. γ values are fixed by the service provider according to the desired QoS.

In the traditional ARC, when inserted, data are always placed in the MRU position in T_1, and when accessed again,

data are promoted to the MRU position in T_2. In the context of QM-ARC, data are treated based on their priority. The data that are labeled high priority follow the same pattern as in ARC. However, if the data are labeled with a lower priority, it is inserted/promoted to a position relative to its priority, between the MRU and LRU positions of T_1 and/or T_2. We use γ_k as a coefficient to find the right position according to the current one if the data are already in the list or according to the beginning of the list. A higher value of γ_k leads to the promotion of data priority k closer to the MRU position. This value equals 1 for highest priority data (promoted to the MRU position). To fall back to traditional ARC, we can fix all values of γ_k to 1, all data get promoted to the MRU position.

This change in the promotion function makes sure that the cache does not get saturated with low-priority data, while still giving them the opportunity to remain in the cache and get promoted if frequently accessed.

IV. Evaluation

A. Methodology

Several metrics were evaluated for a given set of strategies on synthetic and real workloads.

1) Tested metrics and strategies: Three metrics were evaluated for a set of six strategies: i) the penalty cost generated by cache misses per priority level, ii) the global hit ratio per tier, iii) the average hit rate for each priority level per tier. The six strategies tested are : LRU, LFU, Random, Priority-LRU [16], M-ARC and QM-ARC.

We used the very popular LRU and LFU strategies. To compare to state-of-the-art work, we have adapted Priority-LRU [16], the more recent work related to ours, that uses two LRU lists, one per priority level. M-ARC consists of removing the QoS management from QM-ARC, that is managing multi-tier without considering the QoS. Evaluating QM-ARC with M-ARC makes it possible to determine the relevance of considering QoS and the penalty function.

Table I shows the penalty function for each priority level, high P_0 and low P_1 [14]. γ_1 was set to 0.2.

TABLE I: Penalty Function

Delay (ms)	$P_0 \cdot 10^{-8}\$$	$P_1 \cdot 10^{-8}\$$
< 20	0	0
< 150	50	10
$>= 150$	75	15

We varied the cache proportion according to the dataset size to study the efficiency of the approach.

In the experiments performed, we have focused on two-level priority data for simplicity and on a two-tier storage architecture. The first tier is supposed to be a DRAM memory and the second on a high performance SSD (such as the Optane). The cache size is a proportion (%) of the workload size. The size of the DRAM is a fifth of the size of the SSD.

2) Traces: We used both synthetic and real traces that have been widely used to evaluate caching strategies.

Zipf-Like: We use a synthetic trace that follows the Zipf distribution, where the probability of referencing the data i is proportional to $1/i^{\alpha}$. Zipf approximates many common access patterns based on the value of α, like 0.8, corresponding to User Generated Contents (UGC) like YouTube. We created a trace using $\alpha = 0.8$.

IBM: We used the object-store released trace by IBM Research on the SNIA IOTTA repository[1], specifically, the IBMObjectStoreTrace000part0.

Jedi: We generated traces through Jedi[2], a synthetic trace generator that mimics the original traces of Akamai's production CDNs. We chose the class video for our trace.

In all traces, the number of objects is 10 000, the percentage of high-priority content is 20% and the rate of incoming requests is 200 requests per second. Note that cache distribution over several nodes is out of the scope of this paper, as such, only one node was considered.

B. Results and Discussion

Penalty: Figure 2 show the penalty of the six strategies. Overall, QM-ARC reduces the penalty the most compared to other strategies, especially for the IBM trace, up to 83% compared to Priority LRU. In addition, QM-ARC scales well with the growth of the cache proportion, as it enhances penalty in a better way as compared to the other strategies. With the synthetic trace, it decreased by 80%, from 0.5% to 10% proportion of cache, compared to the others.

When analyzing the Jedi trace, we observe that the results of the six strategies are quite similar. However, there is a slight difference between Priority LRU and QM-ARC, with Priority LRU performing 2% better. This divergence can be attributed to the nature of the Jedi traces, which have a longer tail compared to the IBM and synthetic traces. This means that there are more frequently accessed data items. Since 20% of the data have a high priority level, the low priority items tend to remain longer in the cache because they are accessed more frequently. This phenomenon contributes to the slightly better performance of Priority LRU compared to QM-ARC, as the former bounds the position of low priority data while QM-ARC allows them to compete with high-priority ones.

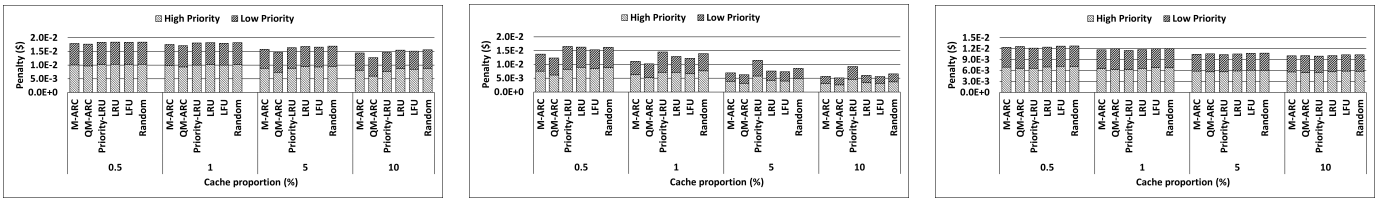
Cache hit Ratio: Figure 3 shows the global cache hit ratio, the high priority data hit ratio and the low priority data hit ratio with the synthetic workload. Overall, we notice that the higher the cache proportion, the higher the hit rate for all strategies, which is a predictable behavior.

For the global hit ratio, M-ARC and QM-ARC gave the best results, an increase of 48% compared with LRU and Priority-LRU. QM-ARC improves the hit rate for the high priority data by 67% compared to Priority LRU and M-ARC and LFU, and by 181% compared to LRU and Random. We notice that for the hit rate of low priority data, M-ARC dethroned QM-ARC, because QM-ARC privileges the high priority data in spite of the low priority ones, which proves the efficiency of the QoS management strategy.

Another interesting result is that QM-ARC takes, the most, advantage of the fastest tier for high priority data. The hit rate

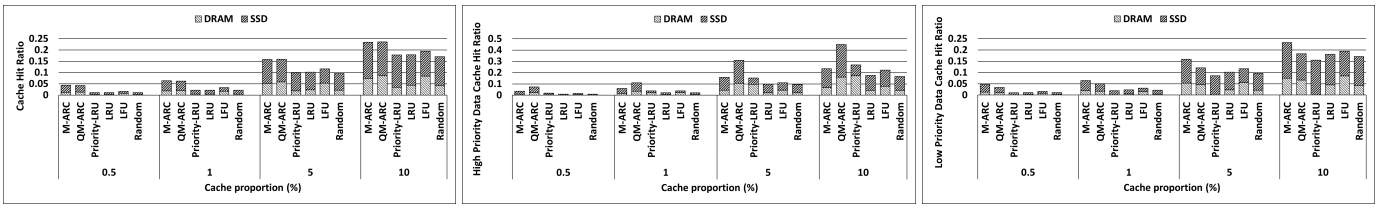
[1] http://iotta.snia.org/

[2] https://github.com/UMass-LIDS/Jedi



(a) Synthetic trace (b) IBM Trace (c) Jedi trace

Fig. 2: Penalty cost



(a) Global cache hit ratio (b) High priority cache hit ratio (c) Low priority cache hit ratio

Fig. 3: Synthetic trace

for low priority data in SSD with Priority LRU is zero because the low priority data never access the DRAM tier as they are only promoted in the SSD.

1) Solution Overhead: ARC policy has a constant-time complexity per request. QM-ARC introduces an additional overhead due to the calculation of two extra parameters. First, a penalty function is used to assess the priority of the data and their position of insertion. Second, the tier sizes are taken into account when changing the size p of the list T_1. The inclusion of these additional factors introduces some negligible overhead in terms of computational complexity and memory usage. The memory usage is negligible, but the computational complexity of the index of insertion is O(n) with n being the number of tiers (which is small enough and gives negligible overhead).

V. Conclusion

In this paper, we designed a generic policy for a multi-tier caching architecture that considers QoS. Our policy is based on ARC, one of the most efficient and popular cache architectures. Our strategy uses the cache tiers in a proportional way to strike a balance between recency and frequency, as in the original ARC. It also borrows the concept of QoS and SLA from the Cloud to implement a cost-benefit strategy in regard to data management for different priority levels. The proposed strategy demonstrated that it can efficiently consider QoS by reducing the penalty compared to other state-of-the-art work. A service provider can fine-tune the penalty function to prioritize certain applications or users. Further work will be achieved by incorporating Reinforcement Learning to consider a deeper history than only immediate past accesses. Another direction we are studying is about distributing multi-tier caches on distant nodes for large systems. In this context, one needs to study how we can make several multi-tier caches work in a collaborative way.

References

[1] M. Soltaniyeh, V. Lagrange Moutinho Dos Reis, M. Bryson, X. Yao, R. P. Martin, and S. Nagarakatte, "Near-storage processing for solid state drive based recommendation inference with smartssds®," in *ACM/SPEC ICPE*, 2022, pp. 177–186.

[2] J. Boukhobza and P. Olivier, *Flash Memory Integration: Performance and Energy Issues*, 1st ed. ISTE Press - Elsevier, 2017.

[3] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao, "Emerging nvm: A survey on architectural integration and research challenges," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 2, nov 2017.

[4] M. Abashkin, A. Natanzon, and E. Bachmat, "Integrated caching and tiering according to use and qos requirements," in *IEEE IPCCC*, 2015, pp. 1–8.

[5] L. V. Yovita and N. R. Syambas, "Caching on named data network: a survey and future research." *International Journal of Electrical & Computer Engineering (2088-8708)*, vol. 8, no. 6, 2018.

[6] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3, no. 2003, 2003, pp. 115–130.

[7] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, "Driving cache replacement with ml-based lecar." in *HotStorage*, 2018, pp. 928–936.

[8] L. V. Rodriguez, F. B. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with cacheus." in *FAST*, 2021, pp. 341–354.

[9] X. Du and C. Li, "Sharc: improving adaptive replacement cache with shadow recency cache management," in *Middleware*, 2021, pp. 119–131.

[10] R. Santana, S. Lyons, R. Koller, R. Rangaswami, and J. Liu, "To arc or not to arc," in *FAST*, 2015, pp. 14–14.

[11] G. Yadgar, M. Factor, K. Li, and A. Schuster, "Mc2: Multiple clients on a multilevel cache," in *ICDCS*, 2008, pp. 722–730.

[12] B. S. Gill, "On multi-level exclusive caching: Offline optimality and why promotions are better than demotions." in *FAST*, vol. 8, 2008, pp. 1–17.

[13] P. Singh and N. Sarma, "Adaptive replacement cache with quality of service for delay sensitive applications in named data networking," in *INDICON*. IEEE, 2021, pp. 1–6.

[14] A. Chikhaoui, L. Lemarchand, K. Boukhalfa, and J. Boukhobza, "Stornir, a multi-objective replica placement strategy for cloud federations," in *ACM SAC*, 2021, pp. 50–59.

[15] L. Ait-Oucheggou, M. I. Naas, Y. Hadjadj-Aoul, and J. Boukhobza, "When iot data meets streaming in the fog," in *IEEE ICFEC*, 2022, pp. 50–57.

[16] M. P. Pamungkas, S. A. Ekawibowo, and N. R. Syambas, "Priority based multilevel cache lru on named data network," in *IEEE ICWT*, 2019, pp. 1–4.