To ARC or not to ARC

Ricardo Santana *Steven Lyons* *Ricardo Koller*[†] *Raju Rangaswami* *Jason Liu*
Florida International University [†]IBM TJ Watson

Abstract

Cache replacement algorithms have focused on managing caches that are in the *datapath*. In datapath caches, every cache miss results in a cache update. Cache updates are expensive because they induce cache insertion and cache eviction overheads which can be detrimental to both cache performance and cache device lifetime. *Non-datapath caches*, such as host-side flash caches, allow the flexibility of not having to update the cache on each miss. We propose *the multi-modal adaptive replacement cache* (mARC), a new cache replacement algorithm that extends the adaptive replacement cache (ARC) algorithm for non-datapath caches. Our initial trace-driven simulation experiments suggest that mARC improves the cache performance over ARC while significantly reducing the number of cache updates for two sets of storage I/O workloads from MSR Cambridge and FIU.

1 Introduction

CPU and main memory caches are *datapath caches*. Such caches incur *forced cache updates*; they are required to make a cache update on every cache miss so that the data is accessible by upper-level hardware or software. The widely used cache replacement policies today, such as (e.g LRU [2], FIFO [2], ARC [8], MQ [13]) were designed for datapath caches. *Non-datapath caches*, on the other hand, are not required to perform a cache update on every cache miss. One can apply *opportunistic cache updates*, whereby case-by-case decisions can be made whether to perform a cache update. A host-side flash cache is an example of a non-datapath cache. Host-side flash caches are attractive because they can reduce the demands placed on network storage, speed up I/O performance, and provide I/O latency and throughput control [1, 6, 7].

A cache update is composed of two operations: an *eviction* and an *insertion*. Performance wise, evictions can be detrimental. In case of a host-side flash cache, a dirty item chosen for eviction needs to be written to the backing storage consuming both cache and network storage bandwidth and contending with other items being accessed at the same time. Furthermore, it can add significant latency to the access inducing the eviction. More significantly, since the item being evicted may be more valuable than the item being inserted in its place, cache updates can be detrimental to the cache hit rate. This problem is especially acute for one-time access items, as in a streaming or random access workload, since they lead to inserting non-reusable items into the cache. Finally, flash-based cache devices have limited write cycles and cache updates also affect device lifetime.

Lazy adaptive replacement cache (LARC) [3] is a recent proposal that implements opportunistic cache updates. While LARC benefits from avoiding certain cache updates, since the LARC cache filter is always operational, it can also prevent important items from entering the cache in a timely fashion. As we shall demonstrate later, because of this shortcoming, LARC performs worse than ARC for the MSR Cambridge workloads.

We propose *multi-modal adaptive replacement cache (mARC)*, a non-datapath version of the adaptive replacement cache (ARC) algorithm. mARC is designed to avoid unnecessary cache updates. It identifies three possible states in which a workload may be operating in at any given time — **STABLE**, **UNSTABLE**, and **UNIQUE ACCESS** — and selectively disables cache replacement depending on the state.

An evaluation of mARC using a cache simulator for the MSR and FIU block I/O traces from SNIA [11] is encouraging. For the MSR Cambridge Traces, while maintaining a competitive hit-rate compared to ARC (1% worse on average), mARC reduces the number of cache updates by 25% on average. This translates to a significant improvement in flash cache device lifetimes. For the FIU traces, mARC leads to 9% better hit-rate on average while reducing the number of cache updates by 23% on average, when compared with ARC. These results motivate further investigation into replacement algorithms that are specifically designed for non-datapath caches.

2 The Case for Selective Caching

2.1 Dynamic Storage Workloads

Storage workloads are dynamic. We model this dynamism using a simple Active-Unique (A-U) model that is time-aware and describes the amount of unique data accessed as well as the amount of data that is reused (active data) in a workload. A sample A-U plot is presented in Figure 1 for one day of the *prn0* MSR Cambridge trace. The A-U model tracks the number of *active* and *unique* pages accessed over time. The active pages at any instant are the unique pages accessed previously and that

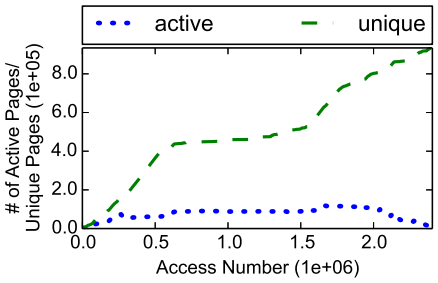

Figure 1: A-U plot for one day of the prn0 MSR trace.

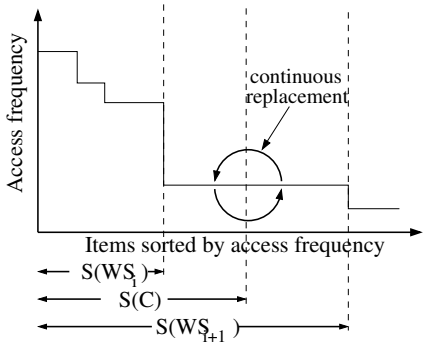

Figure 2: Cache churning behavior. *$S(WS_i)$ is the size of the working set at granularity i that is entirely contained within the larger working set WS_{i+1} [5]. S(C) is the cache size. Since the cache is not large enough to contain all pages of WS_{i+1}, non cache-resident pages of WS_{i+1} continuously replace cache-resident ones.*

will be re-accessed at some time in the future. As pages get accessed, the unique set of pages never decreases, while the active set may either increase or decrease depending on the future reuse.

Several combinations of A-U states can be identified with corresponding workload behavior. We say that a workload is in a **STABLE** state in a given period if the set of items that are currently referenced and their relative importance (e.g., relative frequency of access) remains approximately the same as the previous period. When the set of active and unique pages both remain unchanged, the relative frequency of items being accessed determine if the workload is in a **STABLE** state. When the set of unique pages remains the same but the set of active pages decreases, as well as when both the active and unique page sets increase, the working set is changing, and the workload is an **UNSTABLE** state. When the set of unique pages increases but the set of active pages remains unchanged, one time items (either streaming or random) are being accessed and we refer to the workload as being in the **UNIQUE ACCESS** state.

While the active/unique page states provide a good framework to understand workload behavior, accurately tracking the active/unique page sets and their relative importance is expensive. Further, it is not possible to identify active pages in practice, given that it would require knowledge of future accesses. In practice, we found that the *cache hit rate*, a relatively lightweight metric, can serve as an effective proxy for identifying the above set of workload states and state change events.

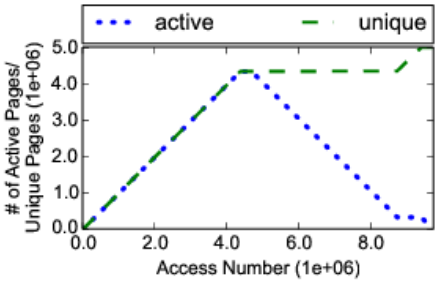

Figure 3: A-U plot for one day of src2-1 MSR trace.

2.2 The Deceptiveness of Stability

If the workload is in a **STABLE** state, a cache is expected to perform well. However, if the cache is not large enough to contain the current set of active pages, the cache contents can *churn* needlessly due to forced cache updates in conventional caching algorithms. Such churning involves the constant eviction of cached pages when pages with relatively equal or lesser importance get accessed. Figure 2 illustrates cache *churning* behavior within the **STABLE** state. Cache replacement due to such churning is detrimental to cache hit rate. The optimal operation in such situations is a "noop", i.e., not to perform cache replacement. This phenomenon has been observed for CPU caches earlier [10].

2.3 ARChilles' Heel

ARC [8] is a high-performance algorithm for datapath caches. During **UNSTABLE** periods, ARC updates the cache contents efficiently by quickly detecting the important items in the new working set. During **UNIQUE ACCESS** periods, ARC retains the frequently used items while unique items tracked in ARC's T1 list pass through the cache quickly. During a workload's **STABLE** state, ARC is designed to retain frequently used items from its T2 list in the cache. However, ARC's cache updating behavior can compromise cache hit ratio when a stable working set does not fit in the cache. Relatively less frequently used items continuously lead to cache misses and cache contents churn needlessly.

Figure 3 presents an A-U plot for src2-1 MSR Cambridge trace. In this workload, the number of active and unique pages climbs steadily (introducing a new working set) until about 4.2M accesses. Following this, the workload reuses a subset of these pages for about 0.2M accesses and then accesses a majority of the 4.2M unique pages exactly once again causing a steady decrease in

Algorithm	STABLE (no Churning)	STABLE (Churning)	UNSTABLE	UNIQUE ACCESS
ARC	✓	∅	✓	∼
LARC	✓	∼	∼	✓
mARC	✓	✓	✓	✓

Legend: ✓ full support, ∼ partial support, ∅ no support

Table 1: Algorithms and workload states

active pages while the number of unique pages stay the same. If the cache size is smaller than the maximum number of active pages (i.e., smaller than 20GB), ARC would continuously evict pages that are about to be used in this trace. This churning would result in a significantly increased cache miss rate.

2.4 Avoiding Forced Cache Updates

The recently proposed LARC [3] algorithm implements a filtering mechanism that *always* avoids cache updates for items not accessed sufficiently recently. LARC consists of two LRU lists, one for cached items and a *filter list* for tracking items that were not in the cache or in the filter list when last accessed. A cache update is only performed when an item that is not found in the cache *is* found in the filter list. The filter list size grows (*resp.* shrinks) with a decrease (*resp.* increase) in cache hit rate.

While LARC presents a new approach for avoiding cache updates, it has its own set of weaknesses. LARC's filter is always operational and can prevent important items from entering the cache in a timely fashion. More specifically, LARC populates the cache at least twice as slowly as other algorithms when workload working sets change, negatively affecting cache performance. In such situations, LARC can perform significantly worse than ARC. LARC's filtering mechanism can reduce the probability of content churning in the cache by shrinking the size of the filter. However, it can only be successful in doing so when the cache hit rate is sufficiently high; in other cases, LARC is unable to avoid churning.

3 mARC

Building on the strengths of ARC and LARC while addressing their weaknesses can lead to significant benefits for non-datapath caches. The intuition behind mARC is that workloads comprise of multiple states and filtering (or *not* filtering) exclusively is not effective in every workload state.

3.1 Design

mARC characterizes workloads as a state machine with three states — **STABLE**, **UNSTABLE**, and **UNIQUE ACCESS**. By avoiding cache updates when cache contents would otherwise churn, mARC improves cache hit-rate. By avoiding unnecessary cache updates, it also improves data access latency and extends cache device lifetime.

During the **UNSTABLE** state, mARC implements a simple ARC access for each referenced item, i.e., performs no filtering. mARC implements *filtering* in the **UNIQUE ACCESS** and **STABLE** states. In these states, items either enter the cache or are registered in a *filter list*, which only stores metadata about the item. On a cache miss, items that are not found in the filter list get added to the filter list, whereas those that are in found in the filter list result in an ARC access within the cache. The size of the filter list is maintained and updated as in LARC [3]. Table 1 provides a qualitative comparison of ARC, LARC, and mARC.

3.2 The States of mARC

To efficiently identify workload states, mARC uses sampling. It maintains two indicators to track workload state: a running average cache hit-rate that has been accumulated during the current workload state (HR_{state}) and the cache hit-rate during the current *sample period* (HR_{sample}). HR_{sample} is the hit-rate computed over the last n accesses where n is the size of the cache. In practice, n accesses provide us a valuable mean hit-rate by ensuring adequate coverage of items compared to the working set resident in the cache.

mARC operates as follows. Every n accesses, $HR_{sample}$is compared with HR_{state}. mARC resets the value of HR_{state} only when entering an **UNSTABLE** state to quickly and accurately track the hit-rate of a new working set. Doing so for the other states is not necessary. While in the **UNSTABLE** state, to increase confidence and robustness, HR_{state} tracks at least $2n$ accesses before checking if a change of state should take place. Table 2 depicts the state machine that mARC implements. The constants used for each condition were determined by experimenting with a subset of the possible combinations of feasible values across all the 45 I/O workloads from MSR Cambridge and FIU [11]. mARC starts its operation in the **UNSTABLE** state. The rest of this section discusses how the various states of mARC operate, how state transitions occur, and the resulting impact to the caching mechanism.

3.2.1 STABLE State

At the beginning of a stable state, mARC enables the filtering mechanism, configuring the filter to its minimum size. ARC, active during the previous **UNSTABLE** state, is expected to have populated the cache with the workload's working-set. Filtering in this state prevents cache pollution and needless cache updates. Starting with the minimum sized filter reduces the probability of cache churning as well. When signs of workload instability appear, the filtering mechanism is stopped and mARC returns to the **UNSTABLE** state to repopulate the cache.

The only condition needed for mARC to transition

State	Condition	Action
STABLE	$HR_{sample} <= 0.7 * HR_{state}$	Switch to **UNSTABLE**
UNSTABLE	$0.9 * HR_{state} <= HR_{sample} <= 1.1 * HR_{state}$	Switch to **STABLE**
	$HR_{sample} >= 1.2 * HR_{state} \wedge HR_{sample} > 0.2$	
	$0.5 * HR_{state} > HR_{sample} \vee HR_{sample} < 0.1$	Switch to **UNIQUE ACCESS**
UNIQUE ACCESS	$0.10 * HR_{sample} >$ Filter-$HR_{sample} \vee HR_{sample} > 0.1$	Switch to **UNSTABLE**

Table 2: mARC State transition table

out of the **STABLE** state is that the performance of the cache is deteriorating over time because a new working set. As shown in Table 2, we used a HR_{sample} that was 70% of the HR_{state} as a robust indicator of instability. In practice, we found that higher values make mARC prematurely enter the **UNSTABLE** state for a substantial fraction of the workloads.

3.2.2 UNSTABLE State

In an **UNSTABLE** state, mARC uses ARC without filtering with the objective of populating a new working set. As shown in Table 2, there are three possible conditions that lead to transitions out of this state:

- HR_{sample} is similar to HR_{state} — within 10% of each other. This implies hit-rate stability and mARC infers that the workload itself is stable. To respond, mARC moves to the **STABLE** state to improve the hit-rate by avoiding cache churning and cache updates. We found 10% of HR_{state} to be an acceptable margin of error when detecting stability. If a more rigorous measure of stability were to be used, mARC delays entering the **STABLE** state and in starting to filter unwanted items. If a less rigorous measure were to be used, the risk of changing state prematurely, and thereby compromising cache hit-rate, increases.
- If HR_{sample} is significantly higher than HR_{state} and also above a threshold, mARC determines that recent performance is better than the historical performance in the current state and moves to the **STABLE** state. When HR_{sample} is 20% higher HR_{state}, then the new state is considered significantly better. However, a minimum threshold must also be met by HR_{sample} for this increase to avoid really low values.
- When HR_{sample} is significantly lower than HR_{state} or HR_{sample} is below a minimum value, mARC infers streaming or random (one time) access behavior, and transitions to the **UNIQUE ACCESS** state. mARC employs 50% decrease in hit-rate as a good indicator of unique access behavior and this worked well in practice. Furthermore, if HR_{sample} is below 10%, unique access behavior is automatically inferred.

3.2.3 UNIQUE ACCESS State

In a **UNIQUE ACCESS** state, mARC turns on filtering mechanisms to avoid cache pollution due to cache updates and stops filtering only after transitioning out to an **UNSTABLE** state. Table 2 shows the conditions that bring mARC back to the **UNSTABLE** state. These involve detecting either that (i) a new working set is being introduced or (ii) unique access behavior has terminated. To determine if a new working set is being accessed, mARC samples the hit-rate of items in the filter (Filter-HR_{sample}). If this value exceeds 10%, it concludes that unique access behavior will soon terminate. mARC also uses a minimum HR_{sample} threshold to detect that the **UNIQUE ACCESS** state has terminated and a new working set may already be cache resident. In this case, switching to the **UNSTABLE** state will allow confirming that the working-set is indeed cache resident and eventually switch to the **STABLE** state.

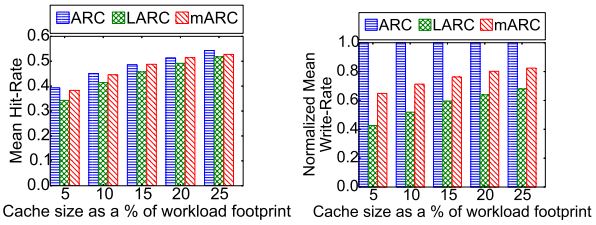

Figure 4: MSR Cambridge Results

4 Evaluation

We built ARC, LARC, and mARC cache simulators which process a block I/O trace and report on the number of reads/writes, hits and misses, and clean and dirty evictions. To simulate sufficient cache as well as I/O activity, we controlled the size of the cache to be a fraction of the *workload footprint*, defined as the combined size of all unique data accessed. We varied this fractional cache size from 5% to 25% in our simulations. Our evaluation metrics include the mean cache hit-rate and the normalized mean write-rate; greater write-rates indicate lower flash cache device lifetime. For the workloads, we used the FIU and MSR Cambridge block I/O traces from the SNIA IOTTA trace repository [11]. The MSR Cambridge and FIU traces are two large sets of 36 and 9 I/O traces respectively, from a variety of production servers/systems [9, 12].

4.1 MSR Traces

We first evaluate mARC for the MSR Cambridge traces, averaging across all its workloads. Figure 4 depicts how

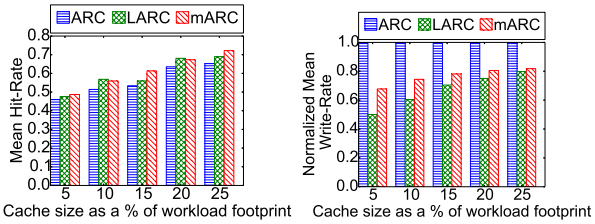

Figure 5: FIU Results

the cache hit-rate and cache write-rate vary for various cache sizes chosen as fractions of the workload footprint size. We observe that mARC has an average hit-rate that is very competitive with ARC (1% worse on average) and is greater than the LARC hit-rate (5% better on average) across the various cache sizes. LARC incurs the least number of cache writes (43% better than ARC on average), while mARC does 25% fewer cache writes on average than ARC.

4.2 FIU Traces

Figure 5 depicts results for the FIU traces, averaged across all the workloads. While mARC and LARC both provide higher hit-rate than ARC, LARC's hit-rate degrades as cache size increases from 10% to 15% of the workload footprint. This happens because LARC makes decisions on what to cache based on a secondary hit in the filter; since the filter size is proportional to the cache size, cache churning becomes a possibility. mARC does slightly better than LARC on average (1% better hit-rate) and also performs better with more cache space. LARC incurs the lowest write-rate (33% lower than ARC), whereas mARC also reduces write-rate by 23% compared to ARC.

5 Related Work

Besides LARC, PLC-Cache [4] also avoids cache updates; it is built on top of a deduplication framework and filters elements based on the frequency of access. Similar to LARC, it does not model workload state and cannot selectively turn off its filtering mechanism based on such state. Some work on CPU caching is also related. Dynamic Insertion Policy (DIP) is an adaptive insertion policy [10] that chooses between using a LRU Insertion Policy (LIP) or a Bimodal Insertion Policy (BIP), based on the current workload. Unlike mARC, the goal of DIP is not to avoid cache updates. Like mARC, it does attempt to reduce overall cache pollution.

6 Conclusion

Conventional datapath caches have been managed by policies that incur forced cache updates. For non-datapath caches, these caching policies can become detrimental to cache performance and cache device lifetime. In this paper, we demonstrate that managing datapath caches involves deciding whether and when to filter items from entering the cache, as well as to turn off such filtering when it becomes detrimental to performance. We show that for caching purposes, workloads can be modeled as a simple but useful state machine. mARC modifies the ARC algorithm equaling or exceeding its hit-rate while significantly lowering cache updates. While much work remains, mARC and other algorithms in its class present a path towards better managing non-datapath caches.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Ashvin Goel, for their insightful feedback. This work is supported in part by NSF awards CNS-1018262 and CNS-1448747, and a NetApp Faculty Fellowship.

References

[1] S. Byan, J. Letini, J. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. *In Proc. of IEEE MSST*, April 2012.

[2] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proc. of ACM SIGMETRICS*, 1990.

[3] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. *Proc. of MSST*, May 2013.

[4] L. Jian, C. Yunpeng, Q. Xiao, and X. Yuan. PLC-Cache: Endurable SSD cache for deduplication-based primary storage. In *Proc. of MSST*, May 2014.

[5] R. Koller, A. Verma, and R. Rangaswami. Generalized ERSS tree model: Revisiting working sets. *Perform. Eval.*, 2010.

[6] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proc. of USENIX FAST*, February 2013.

[7] R. Koller, A. J. Mashtizadeh, and R. Rangaswami. Centaur: Host-side SSD caching for storage performance control. In *Proc. of ICAC*, July 2015.

[8] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. of USENIX FAST*, March 2003.

[9] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. In *Proc. of USENIX FAST*, 2008.

[10] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proc. of ISCA*, 2007.

[11] SNIA. Block I/O traces, 2011. URL `http://iotta.snia.org/tracetypes/3`.

[12] A. Verma, R. Koller, L. Useche, and R. Rangaswami. SRCMap: Energy proportional storage using dynamic consolidation. In *Proc. of USENIX FAST*, February 2010.

[13] Y. Zhou and J. F. Philbin. The multi-queue replacement algorithm for second level buffer caches. USENIX ATC 2001, pages 91–104, June 2001.