H-ARC: A Non-Volatile Memory Based Cache Policy for Solid State Drives

Ziqi Fan, and David H.C. Du
University of Minnesota-Twin Cities
{fan, du}@cs.umn.edu

Doug Voigt
HP Storage
doug.voigt@hp.com

***Abstract*—With the rapid development of new types of non-volatile memory (NVM), one of these technologies may replace DRAM as the main memory in the near future. Some drawbacks of DRAM, such as data loss due to power failure or a system crash can be remedied by NVM's non-volatile nature. In the meantime, solid state drives (SSDs) are becoming widely deployed as storage devices for faster random access speed compared with traditional hard disk drives (HDDs). For applications demanding higher reliability and better performance, using NVM as the main memory and SSDs as storage devices becomes a promising architecture.**

Although SSDs have better performance than HDDs, SSDs cannot support in-place updates (i.e., an erase operation has to be performed before a page can be updated) and suffer from a low endurance problem that each unit will wear out after certain number of erase operations. In an NVM based main memory, any updated pages called dirty pages can be kept longer without the urgent need to be flushed to SSDs. This difference opens an opportunity to design new cache policies that help extend the lifespan of SSDs by wisely choosing cache eviction victims to decrease storage write traffic. However, it is very challenging to design a policy that can also increase the cache hit ratio for better system performance.

Most existing DRAM-based cache policies have mainly concentrated on the recency or frequency status of a page. On the other hand, most existing NVM-based cache policies have mainly focused on the dirty or clean status of a page. In this paper, by extending the concept of the Adaptive Replacement Cache (ARC), we propose a Hierarchical Adaptive Replacement Cache (H-ARC) policy that considers all four factors of a page's status: dirty, clean, recency, and frequency. Specifically, at the higher level, H-ARC adaptively splits the whole cache space into a dirty-page cache and a clean-page cache. At the lower level, inside the dirty-page cache and the clean-page cache, H-ARC splits them into a recency-page cache and a frequency-page cache separately. During the page eviction process, all parts of the cache will be balanced towards to their desired sizes.

I. Introduction

Dynamic random-access memory (DRAM) is the most common technology used for the main memory. Despite DRAM's advantages of high endurance and fast read/write access speed, DRAM suffers from data loss in the event of power failure or a system crash. To solve this problem, combining DRAM's fast access speed and Flash's persistence together, non-volatile DIMMs [1] provide reliable main memory systems. In addition, new types of non-volatile memory (NVM), such as phase change memory (PCM), Memristor and SST-RAM, have rapidly developed into possible candidates for the main memory in future computer systems. These emerging NVM technologies may offer other advantages in addition to their non-volatile nature. For examples, compared with DRAM, Memristor and PCM can achieve higher density, and Memristor and STT-RAM can have faster read accesses and lower energy consumption [5] [6].

Compared with traditional hard disk drives (HDDs), flash-based solid state drives (SSDs) can achieve much faster random access speed and have been widely deployed in small devices such as cameras, cellular phones as well as computer storage devices. Among them, Figure 1 is the architecture that we have adopted throughout this paper. We assume a computer system has a CPU at the top, NVM as the main memory and SSDs as storage devices [7].

SSDs support read/write by pages. A page can only be written to a free page (i.e., a page from a block which is erased), because SSDs do not support in-place updates. Due to the slow speed of erase operation, which takes around 2 ms, the alternative way of writing to an existing page is writing to a free page and marking the original page as invalid. Then a garbage collection mechanism will be triggered periodically or on-demand to reclaim those invalid pages. However, under current technologies, SSDs still suffer from the low endurance problem, especially for MLC flash based SSDs, whose expected erase count is around 1K before wearing out. Secondly, SSDs' write speed (around 200 μs) is much slower than read speed (around 25 μs). Thus, our focus is designing a smart cache policy that: (1) decreases the write traffic from the main memory to storage to extend SSDs' lifespan and shorten SSDs' write processing time; and (2) maintains or even increases the main memory's cache hit ratio for better system performance.

Many existing cache schemes on DRAM-based main memory system mainly concentrate on improving cache read hit ratio for clean pages, because dirty pages (newly written or updated pages) will be flushed back to storage quite frequently for the sake of reliability. With NVM as the main memory, cached dirty pages will not be lost in the occurrence of power failure or a system crash. As a result, the frequency of dirty page synchronization from memory to storage can be reduced dramatically without jeopardizing data consistency [9]. This difference opens an opportunity for manipulating cached dirty pages to decrease write traffic. On the other hand, some space in memory has to be reserved for the read cache (clean pages)

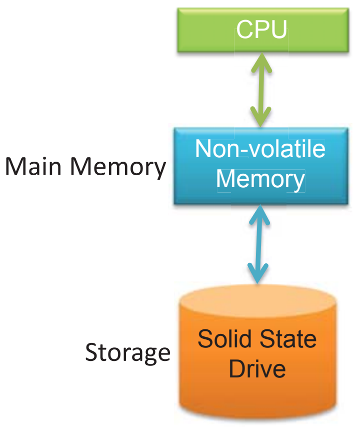


Fig. 1. System Architecture

if system performance is critical for certain applications. Throughout this paper, dirty pages are cached memory pages which have been modified or newly written to memory, but not yet written to the storage device (synchronized). Clean pages are memory pages which have copies of the same data in the storage device.

Due to the limited capacity of main memory, only a certain number of pages can be cached. When the memory is full, a victim page has to be evicted to reclaim space for each newly inserted page. To decrease storage write traffic, one approach is to hold dirty pages at memory as long as possible to delay eviction. As a result, the cache should store as many dirty pages as possible. However, this decision might hurt cache hit ratio, especially cache read hit ratio. Thus, it is challenging to decide the proper cache size for dirty pages under various workload. To solve this problem, an intelligent mechanism should be designed to dynamically split the cache for dirty pages and clean pages. Note that we will always cache a write to reduce the access time of writes. Write request hits on a dirty page will reduce the write traffic to storage. Read request hits on either a clean or dirty page will improve the performance of reads. Therefore, the overall hit ratio is important. Assuming we get perfect cache sizes for dirty pages and clean pages, when the cache is full, it is still non-trivial to decide which victim page should be evicted. Thus, another intelligent mechanism should be designed to help make eviction decisions.

One of the proposed cache policies trying to increase cache hit ratio, named Adaptive Replacement Cache (ARC) [10], faces the similar challenge: under unpredictable workload, how to decide the proper cache sizes for recently used pages and frequently used pages. To deal with the challenge, they seek help from history by maintaining two ghost caches: one for the recency cache and the other for the frequency cache. The recency cache stores pages being referenced once recently and the frequency cache stores pages being referenced at least twice recently. A ghost cache is a data structure storing only the page numbers of recent evicted pages. Each cache can grow or shrink along with the workload's tendency towards recency or frequency. For example, if a ghost cache hit happens in the recency ghost cache, the size of the recency cache will be enlarged, and the size of frequency cache will be shrunk accordingly. Through evaluation, the learning rule behind ARC is proved quite effective and increases cache hit ratio significantly. However, ARC mainly tries to increase cache hit ratio and has not considered how to decrease storage write traffic.

To deal with the above-mentioned challenges, in this paper, by using the learning process of ARC in a hierarchical manner, we propose a Hierarchical Adaptive Cache Replacement (H-ARC) policy that considers all four factors of a page's status: dirty, clean, recency, and frequency. The desired cache sizes for dirty pages and clean pages are first determined at a higher level. Their sizes will be dynamically adjusted based on a similar mechanism as ARC. At the next level, to decide which page to evict, we propose to adopt ARC twice: one for cached clean pages and the other for cached dirty pages. All these cache sizes will be balanced along with the incoming workload and no tuning parameters are needed. For example, if H-ARC detects that the current workload is write intensive, it will enlarge the portion for dirty pages. In addition, if it detects frequency inside the dirty page cache is more popular than recency, the portion for frequency will be enlarged.

Specifically, in the first ARC-like strategy process, we split the whole cache into a dirty page cache portion and a clean page cache portion. If a cache hit happens in the corresponding ghost cache of the dirty page cache (resp. the clean page cache), which is actually a real cache miss, we will enlarge the desired dirty page cache size (resp. the desired clean page cache size). Note that due to fixed total cache size, the other cache's size will be reduced accordingly. To keep dirty pages in the cache longer, we prioritize the desired dirty page cache by enlarging it much faster than the desired clean page cache.

Once the desired size for dirty pages (or clean pages) is decided, the ARC-like strategy is applied again based on receny and frequency inside the dirty page cache and the clean page cache. We will enlarge the desired real cache size - the dirty recency cache or the dirty frequency cache (resp. the clean recency cache or the clean frequency cache), if the cache hit happens in its corresponding ghost cache. The dirty recency cache stores pages that are dirty and referenced once recently and the dirty frequency cache stores pages that are dirty and referenced at least twice recently. Similarly, the clean recency cache and the clean frequency cache are defined. Different from the size changing of the dirty page cache and the clean page cache, we treat the dirty frequency cache and the dirty recency cache (or the clean frequency cache and the clean recency cache) equally by enlarging or shrinking them in symmetric steps. When the cache is full, we will evict a page from a real cache so that all the cache sizes will be balanced towards their desired sizes.

The rest of the paper is organized as follows. In the next section we will discuss related work on cache policies and the differences between theirs and ours. Section III gives a detailed description of our proposed cache policy along with some discussion about system crash recovery. In Section IV, we evaluate the effectiveness of our scheme through various traces and discuss the experimental results. Some conclusions are offered in Section V.

II. Related Work

A. Non-volatile Memory

Due to flash memory's merits of small size, light weight, low power consumption, high shock resistance, and fast random read performance, it has already become the primary non-volatile data storage medium for mobile devices, such as cellular phones, digital cameras and sensor devices [17], [20]. Recently, the popularity of flash memory has also extended from embedded devices to laptops, PCs and enterprise-class servers with flash-based SSDs being widely considered as a replacement for magnetic disks. However, average flash memory has 100x read latency of DRAM's and 1000x write latency of DRAM's without mentioning flash memory's most time consuming operation - erase. In addition, even with the help of proposed wear leveling and garbage collection mechanisms [17], [18], [19], [20], [21], [22], [23], [24], flash memory still has a quite short expected lifetime before it wears out. Therefore, it is unlikely to be a replacement for DRAM as main memory although recently Micron Technology has announced a flash memory based board that can directly connect to DDR3 [25].

With the rapid development of new types of non-volatile memory, these emerging technologies have become promising main memory replacements for DRAM. Phase-change memory (also known as PCM or PRAM), is one of the most promising new type non-volatile memory. PCM exploits the unique behavior of chalcogenide glass to switch the material between two states. The state of a PCM device is changed by heating. Different heat-time profiles are used to switch from one phase to another. PCM can give higher scalability and storage density than DRAM. A larger memory can significantly reduce page faults and disk I/O requests. Since PCM is non-volatile in contrast to DRAM, using PCM as the primary memory will have additional capabilities, such as fast wakeup, low overhead checkpointing and restart for HPC applications. Clearly, a PCM-based memory hierarchy can present density, scalability, and even power saving advantages. In general, PCM still has 5 to 10 times higher latency than DRAM. To overcome PCM's speed deficiency, different system architectures have been designed to integrate PCM into current system without performance degrading [5], [26], [27], [28], [29], [30], [31]. As another front runner of new type non-volatile memory, STT-RAM explores spin-transfer torque technology [32]. Its operation is based on magnetic properties of special materials whose magnetic orientation can be controlled and sensed using electrical signals. The Memristor, short for memory resistor - is another class of electrical circuit and a strong non-volatile memory competitor under fast development by HP [33].

B. Cache Policies

Most existing work on DRAM-based main memory systems mainly concentrates on improving cache read hit ratio since dirty pages will be flushed back to storage quite frequently. Belady's optimal page replacement policy leads to the optimal cache hit ratio [34]. This algorithm always discards pages that will not be needed for the longest time in the future. However, since precisely predicting the future access patterns is impossible, Belady's algorithm is not practical. All cache policies try to improve their cache hit ratio towards Belady's.

To predict a page's future access pattern, recency and frequency are two valuable indicators. Some of the existing work, such as Least Recently Used (LRU) and Least Frequently Used (LFU) [35], only consider one factor and overlook the other one. As an improvement, Adaptive Replacement Cache (ARC) [10] splits the whole cache into two smaller real caches: one recency cache storing pages being referenced once recently and one frequency cache storing pages being referenced at least twice recently. In addition, two ghost caches are maintained, one for each real cache. A ghost cache is a data structure storing only the metadata of recent evicted pages. A ghost cache hit indicates perhaps this type of real cache needs to be enlarged. This is considered as a learning process. Each real cache can grow or shrink along with the workload's tendency to recency or frequency based on the ghost cache hits.

Several existing studies on cache policies for NVM-based main memory systems, similar to the architecture we adopt, focus on decreasing storage write traffic to extend the lifetime of SSDs. For example, Park *et al.* propose a Clean First LRU (CFLRU) algorithm which splits the whole cache into a working region and a clean-first region [36]. The clean-first region is one portion of the cache near the end of LRU position. Clean pages will be evicted first from the clean-first region following an LRU order. Dirty pages will be evicted if no clean page is left in the clean-first region. CFLRU does not bring frequency into consideration and needs to predefine the size of the clean-first region. However, if the size is set too large, cache hit ratio may be hurt because of early eviction of hot clean pages. On the other hand, if the size is set too small, dirty pages may be evicted too early. Qiu *et al.* propose a cache policy in NVMFS [9] which splits the whole cache into two smaller caches - a dirty page cache and a clean page cache. Each cache will grow and shrink based on page hits and no ghost cache is maintained. It also overlooks frequency. Jung *et al.* enhanced LRU algorithm with an add-on page replacement strategy, called Write Sequence Reordering (LRU-WSR). They give dirty pages a second chance before evicting from cache to decrease write traffic. For each dirty page, they add a bit to denote whether it's a hot page or a cold page. Initially, they assume all the dirty pages are hot. If the current eviction victim is dirty and hot, they mark it as a cold page and migrate it to the MRU position. If the current eviction victim is dirty and cold, or is clean, they evict it right away. If a cache hit happens to a dirty page, it will remain hot or change from cold to hot.

The potential issue behind LRU-WSR is whether giving all the dirty pages a second chance is the best choice. For some cold dirty pages, giving a second chance means evicting some hot clean pages, which will hurt the hit ratio.

Different from using NVM as the main memory, some cache studies have investigated how to decrease flash page writes using NVM inside SSDs. Here, NVM works mainly as a write buffer and no read request is cached. Jo *et al.* propose Flash Aware Buffer management (FAB) that clusters pages in the same block and evict the pages in a block with the largest number [37]. If there is a tie, evict the largest recently used cluster. However, FAB only considers cluster size and overlooks recency. Kang *et al.* propose a Coldest and Largest Cluster (CLC) algorithm which combines FAB and LRU. CLC maintains two lists of clustered pages (sequential pages): (1) the size-independent cluster list sorted in LRU fashion to explore temporal locality for hot clusters; (2) the size-dependent cluster list sorted by cluster size to explore spatial locality for cold clusters [38]. Initially, CLC inserts pages in the size-independent list. When the size-independent list is full, CLC moves clusters from the LRU position of the size-independent to the size-dependent list. When the size-dependent list is full, CLC evicts the largest cluster from its tail. Wu *et al.* propose a Block-Page Adaptive Cache (BPAC) inherited CLC approach [39]. BPAC's difference is that it adaptively adjusts the size of each list based on the workload.

III. Our Proposed Approach: Hierarchical-ARC

A. Approach Overview

We use the learning process of ARC in a hierarchical manner in the proposed H-ARC to consider a page's four different types of status: dirty, clean, recency and frequency. The desired cache sizes for dirty pages and clean pages are first determined at a higher level. These sizes will be dynamically adjusted based on a similar mechanism of ARC. At the next level, the dirty page cache and the clean page cache will be maintained by a separated ARC scheme individually. Our focus is on how these three ARC mechanisms interact with each other to reduce write traffic to SSD and to increase the hit ratio for both reads and writes.

Similar to ARC, we define two types of caches: a real cache storing metadata and data of a page and a ghost cache storing only metadata of a page. A corresponding ghost cache is maintained to each type of real caches for the purpose of adaptively changing its size. We split the whole cache into two cache regions: one for dirty pages denoted by D and the other for clean pages denoted by C. Then, inside each region, we split it into a frequency region and a recency region. As a result, four real caches will coexist: a dirty recency cache denoted by D_{1i}, and a dirty frequency cache D_{2i} for the dirty page cache region, a clean recency cache C_{1i} and a clean frequency cache C_{2i} for the clean page cache region.

Another four ghost caches are maintained, namely D_{1o}, D_{2o}, C_{1o}, C_{2o} which are corresponding to real caches D_{1i}, D_{2i}, C_{1i}, C_{2i} respectively. Our cache name notation follows these intuitions: D means dirty, C means clean, subscript 1 means reference of one time, which captures recency, subscript 2 means reference of at least two times, which captures frequency, subscript i means the cached pages are actually in cache, subscript o means the cached pages are actually out of cache and are ghosts. A ghost cache only stores the page identifier (page number) of recent evicted pages from its corresponding real cache. The size of each cache is the number of pages stored in it. If we define the maximum physical cache size (i.e., memory size) to be L, then the sum of all the real caches can never be larger than L and the sum of all the real caches and ghost caches can never be larger than $2 * L$.

For the purpose of adaptively changing cache sizes of dirty pages and clean pages, a learning process similar to ARC is implemented. In this scheme, we conceptually group D_{1i} (C_{1i}) and D_{2i} (C_{2i}) as the real cache for dirty pages denoted by D_i (clean pages denoted by C_i). Similarly their corresponding two ghost caches are grouped together denoted by D_o and C_o. The details of the cache size adjustment can be found later.

Initially, all the real caches and ghost caches are empty. For every read or write request r from the workload, one and only one of the three cases will happen:

- Real cache hit.
- Real cache miss, but ghost cache hit.
- Both real and ghost cache misses.

B. Real Cache Hit

If a page request r is a read request and a cache hit in C_{1i} or C_{2i}, this page is referenced at least twice and remains a clean page, so we migrate it from its original location in either C_{1i} or C_{2i} to the most recently used (MRU) position in C_{2i}. Similarly, if the request r is a read request and a cache hit in D_{1i} or D_{2i}, this page is referenced at least twice and remains a dirty page, so we migrate it from its original location either in D_{1i} or D_{2i} to the MRU position in D_{2i}.

If the request r is a write request and a cache hit in C_{1i} or C_{2i}, this page is referenced at least twice and changed from a clean page to a dirty page, so we migrate it from its original location to the MRU position in D_{2i}. If the request r is a write request and a cache hit in D_{1i} or D_{2i}, this page is referenced at least twice and remains a dirty page, so we migrate it from its original location to the MRU position in D_{2i}. Note that when we count reference times, we consider both reads and writes.

C. Real Cache Miss, Ghost Cache Hit

For the real cache miss and ghost cache hit case, three steps will happen:

- Adjustment of the desired sizes of the real caches in order to capture the current workload's tendency to writes versus reads and frequency versus recency.
- If the cache is full, a page will be evicted from a real cache such that all the real caches sizes will be balanced towards their desired sizes.
- Insert the new page into its corresponding real cache.

First, we will modify the cache sizes adaptively in a hierarchical manner. At the higher level, targeting the whole cache, we need to decide the desired size for D_i denoted as $\hat{D_i}$ and the desired size for C_i denoted as $\hat{C_i}$. Here, we use an integer P to represent the size of $\hat{C_i}$. Again, we use L to denote the physical memory size as used in Section III.A. Thus,

$$\hat{C_i} = P \tag{1}$$

$$\hat{D_i} = L - P \tag{2}$$

At the lower level, targeting the dirty page region (the clean page region), we need to decide the desired size for D_{1i} denoted as $\hat{D_{1i}}$ and D_{2i} denoted as $\hat{D_{2i}}$ (the desired size for C_{1i} denoted as $\hat{C_{1i}}$ and C_{2i} denoted as $\hat{C_{2i}}$). Here, we use two fractions P_C and P_D to denote the desired proportion for $\hat{C_{1i}}$ and $\hat{D_{1i}}$ inside C_i and D_i respectively. The reason we use fractions instead of integers is a fraction can be more precise in regarding to describe the desired sizes since $\hat{C_i}$ and $\hat{D_i}$ are changed dynamically. Note that throughout this paper, if an integer result is needed, we will take the floor function of the equation. The equations are shown below:

$$\hat{C_{1i}} = P_C * \hat{C_i} \tag{3}$$

$$\hat{C_{2i}} = \hat{C_i} - \hat{C_{1i}} \tag{4}$$

$$\hat{D_{1i}} = P_D * \hat{D_i} \tag{5}$$

$$\hat{D_{2i}} = \hat{D_i} - \hat{D_{1i}} \tag{6}$$

At the higher level, if a ghost page hit happens in C_o, it means previously we should not have evicted this clean page out of cache. To remedy this, we will enlarge $\hat{C_i}$. Every time there is a ghost hit at C_o, $\hat{C_i}$ (or P) will be increased by 1. According to Equation (2), $\hat{D_i}$ will be decreased by the same amount. Note that P can never be larger than L. The equation of P adjustment is shown below:

$$P = min\{P + 1, L\} \tag{7}$$

On the other hand, if a ghost hit happens in D_o, it means previously we should not have evicted this dirty page out of cache. To remedy this, we will enlarge D_i. In order to save write traffic and keep dirty pages in the cache longer, different from the increment of $\hat{C_i}$, we enlarge $\hat{D_i}$ much faster. If the size of C_o is smaller than D_o, $\hat{D_i}$ will be increased by two. If the size of C_o is greater than or equal to D_o, $\hat{D_i}$ will be increased by two times the quotient of the cache sizes of C_o and D_o. Thus, the smaller the size of D_o is, the larger the increment is. According to Equation (1), $\hat{C_i}$ will be decreased by the same amount. Again, the combined size of C_i and D_i can never be larger than L and P cannot be smaller than 0. The equation of P adjustment is shown below:

$$P = \begin{cases} max\{P - 2, 0\} & \text{if } |C_o| < |D_o| \\ max\{P - 2 * \frac{|C_o|}{|D_o|}, 0\} & \text{if } |C_o| \geq |D_o| \end{cases} \tag{8}$$

After the higher level adjustment, in the lower level (inside the dirty page region or the clean page region), if a ghost page hit happens in C_{1o} or D_{1o}, it means previously we should not have evicted this recency page out of cache. To remedy this, we will enlarge the proportion for $\hat{C_{1i}}$ or $\hat{D_{1i}}$ by increasing P_C or P_D accordingly. Similarly, if a ghost page hit happens in C_{2o} or D_{2o}, it means previously we should not have evicted this frequency page out of cache. To remedy this, we will enlarge the proportion for $\hat{C_{2i}}$ or $\hat{D_{2i}}$ by decreasing P_C or P_D accordingly. Here, different from the higher level dirty and clean region size adjustment, the adjustment of frequency and recency cache size is symmetric since we have no clear preference. After the adjustment of P_C or P_D, $\hat{C_{1i}}$, $\hat{C_{2i}}$, $\hat{D_{1i}}$ and $\hat{D_{2i}}$ will be recalculated through Equations (3)-(6). The equations of P_C and P_D adjustments are shown below:

- If the ghost page hit happens in C_{1o}, we will enlarge the proportion for $\hat{C_{1i}}$, so P_C will be increased:

$$P_C = \begin{cases} min\{P_C + \frac{1}{P}, 1\} & \text{if } |C_{2o}| < |C_{1o}| \\ min\{P_C + \frac{\frac{|C_{2o}|}{|C_{1o}|}}{P}, 1\} & \text{if } |C_{2o}| \geq |C_{1o}| \end{cases} \tag{9}$$

- If the ghost page hit happens in C_{2o}, we will enlarge the proportion for $\hat{C_{2i}}$, so P_C will be decreased:

$$P_C = \begin{cases} max\{P_C - \frac{1}{P}, 0\} & \text{if } |C_{1o}| < |C_{2o}| \\ max\{P_C - \frac{\frac{|C_{1o}|}{|C_{2o}|}}{P}, 0\} & \text{if } |C_{1o}| \geq |C_{2o}| \end{cases} \tag{10}$$

- If the ghost page hit happens in D_{1o}, we will enlarge the proportion for $\hat{D_{1i}}$, so P_D will be increased:

$$P_D = \begin{cases} min\{P_D + \frac{1}{L-P}, 1\} & \text{if } |D_{2o}| < |D_{1o}| \\ min\{P_D + \frac{\frac{|D_{2o}|}{|D_{1o}|}}{L-P}, 1\} & \text{if } |D_{2o}| \geq |D_{1o}| \end{cases} \tag{11}$$

- If the ghost page hit happens in D_{2o}, we will enlarge the proportion for $\hat{D_{2i}}$, so P_D will be decreased:

$$P_D = \begin{cases} max\{P_D - \frac{1}{L-P}, 0\} & \text{if } |D_{1o}| < |D_{2o}| \\ max\{P_D - \frac{\frac{|D_{1o}|}{|D_{2o}|}}{L-P}, 0\} & \text{if } |D_{1o}| \geq |D_{2o}| \end{cases} \tag{12}$$

After all the desired cache size adjustments, we call the Eviction&Balance (EB) algorithm to evict a real page if the real caches are full. Note that if the real caches are not full, all the ghost caches will be empty since a ghost page will be generated only after a real page eviction. The EB algorithm will be introduced in Subsection III.E.

Finally, we will insert the page into the MRU position of C_{2i} if it's a read request and D_{2i} if it's a write request. Clearly, the hit ghost page will be deleted.

D. Both Real and Ghost Cache Misses

The last case is a request r misses in both real caches and ghost caches. When the real caches are not full, we can simply insert the page into the MRU position of C_{1i} if r is a read request, or into the MRU position of D_{1i} if r is a write request.

When the real caches are full, we need to evict a real page out of cache to reclaim space for the new page insertion. At the same time, we try to equalize the size of D and C, which are towards L for both of them. In addition, inside D we try to equalize the size of D_1 and D_2, and the same way for C. Specifically, D includes D_{1i}, D_{2i}, D_{1o} and D_{2o}. C includes C_{1i}, C_{2i}, C_{1o} and C_{2o}. D_1 includes D_{1i} and D_{1o}. D_2 includes D_{2i} and D_{2o}. C_1 includes C_{1i} and C_{1o}. C_2 includes C_{2i} and C_{2o}. The reason of this equalization is that we want to avoid "cache starvation". "Cache starvation" can happen in H-ARC if one real cache size and its corresponding ghost cache size are both very large. Because the sum of all the cache sizes are fixed, the other real cache size must be very small as well as its corresponding ghost cache size. In this situation, the side with the smaller ghost cache size has difficulty growing bigger in a short duration, even if the current workload favors it, since fewer ghost cache hits can happen.

To achieve the goal of equalization, we will check the size of C. If the size of C is greater than L, which means it already takes more than half of the total cache space including both real and ghost caches, then we will evict from this side. Otherwise, we will evict from D. Assuming we decide to evict from C, inside C, we will check the size of C_1. If the size of C_1 is greater than $L/2$, which means it already takes half of the total cache space for C, we will evict from C_1. Otherwise, we will evict from C_2. The eviction process in D is similar to the process in C.

When we actually perform an eviction from a region, e.g. C_1, we will evict the LRU page in C_{1o} and call EB algorithm to identify and evict a real page. The reason for evicting a ghost page out first is when the EB algorithm evicts a real page, this page needs to be inserted into its corresponding ghost cache. However, if C_{1o} is empty, we have no choice but to evict the LRU page in C_{1i}.

Finally, after a real page eviction, a free slot is reclaimed and we can insert the new page into the MRU position of C_{1i} if r is a read request, or into the MRU position of D_{1i} if r is a write request.

E. Eviction&Balance (EB) Algorthim

In the last two cases, a new page needs to be inserted into the real cache. In case the real caches are full, we need to evict a page out of cache to reclaim space for this new page. We design an Eviction&Balance (EB) algorithm to identify a real page to be evicted and to balance the real cache sizes towards their desired sizes. With the defined P, P_D and P_C, we can easily calculate the desired size of C_i, D_i, C_{1i}, C_{2i}, D_{1i}, D_{2i} though Equations (1)-(6). After obtaining all the desired sizes, we compare them with the current size of each real cache. We will evict from one real cache that is larger than its desired size.

Specifically, at the higher level, if the size of C_i is larger than or equal to $\hat{C_i}$ and the request r is in D_o, we will evict a page from C_i. Otherwise, we will evict a page from D_i. At the lower level assuming we are evicting from C_i, if the size of C_{1i} is larger than $\hat{C_{1i}}$, we will evict the LRU page out from C_{1i} and insert its page number into the MRU position in C_{1o}. Otherwise, we will evict the LRU page out from C_{2i} and insert its page number into the MRU position in C_{2o}. Similar operation will happen in D_i if we need to evict a page out from this side.

F. System Consistency and Crash Recovery

System crashes are inevitable, hence it is always an important issue in designing a consistent system that can recover quickly. Since H-ARC chooses to delay dirty page synchronization, the chance of many dirty pages staying in the cache after system crashes will be high. Here, we propose two simple solutions facing two different kinds of system failures.

When facing system crashes or unexpected power failures, we have to reboot the whole system. In order to make sure the system is consistent, all the dirty pages will be flushed back through the following steps. The boot code needs to be modified such that the page table will be well retained in NVM regardless of the crashes. Then, identify all the cached dirty pages from the page table, and synchronize them to the storage immediately. Finally, reinitialize the page table and continue the regular booting process.

When facing hardware failures, the dirty pages in NVM may not be recoverable. To mitigate the risk of losing data, we add a timer to each dirty page, such that a dirty page must be flushed back after certain time elapses. For example, after a page is updated for one hour, it will be forced written to the storage and become a clean page. In H-ARC, this page will be migrated from D_{1i} to C_{1i}, or from D_{2i} to C_{2i}.

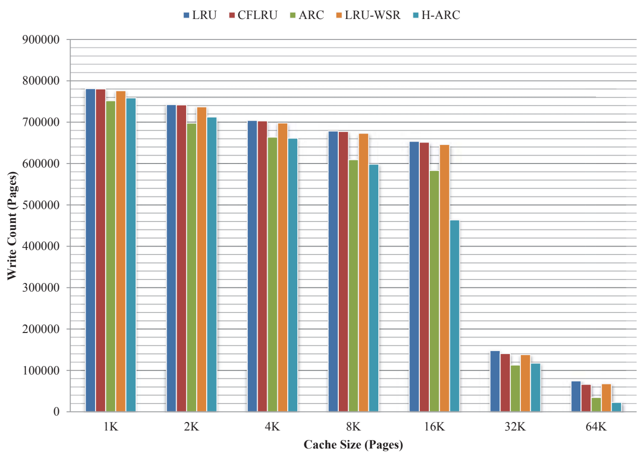
Trace Name	Total Requests	Unique Pages	Read/Write Ratio
mds_0	11,921,428	741,522	1:2.56
wdev_0	2,368,194	128,870	1:3.73
web_0	7,129,953	1,724,201	1:0.76
fio zipf	524,411	291,812	1:0.25
fio pareto	524,345	331,137	1:0.25
File server	1,417,814	406,214	1:0.35

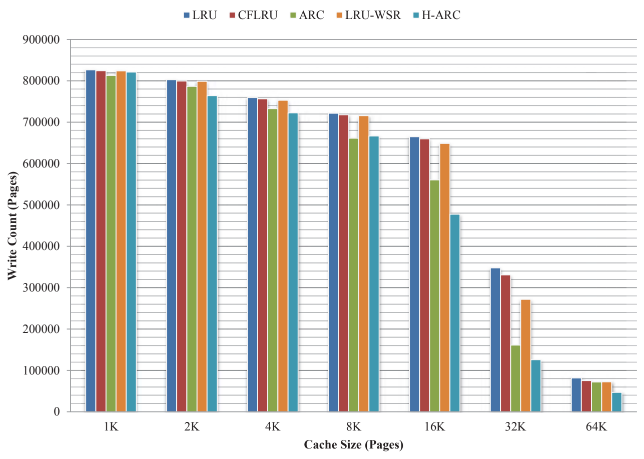
TABLE I
A SUMMARY OF TRACES USED IN THIS PAPER.

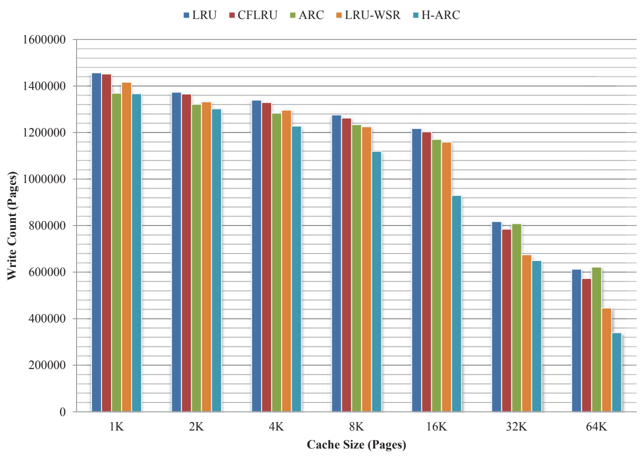
IV. Evaluation

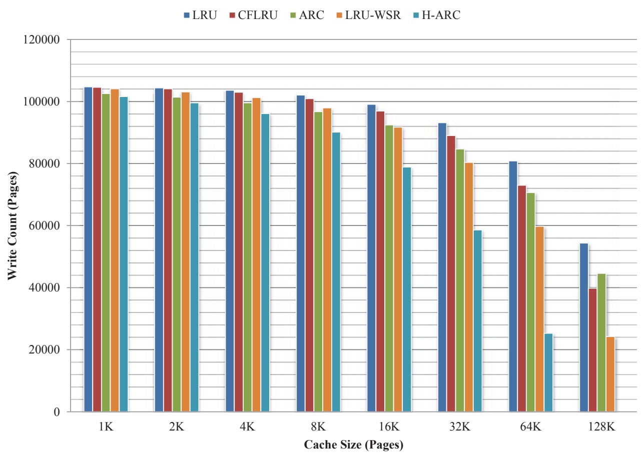
In this section, we evaluate our proposed cache policy along with several existing ones as listed below (detailed algorithm description can be found in Section II.B):

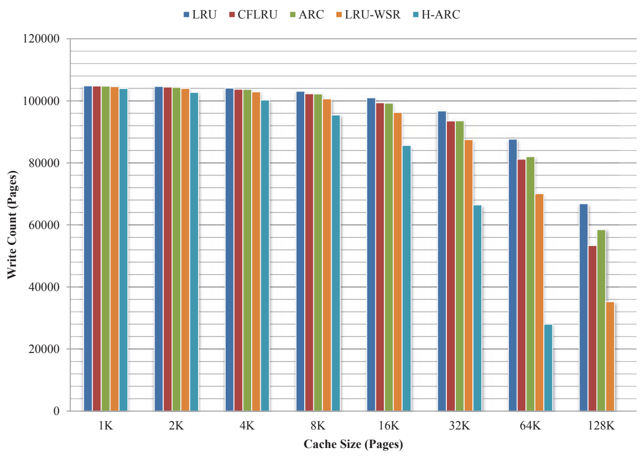
- *MIN*: Belady MIN, optimal offline cache policy.
- *LRU*: Least Recently Used cache policy.
- *CFLRU*: Clean First LRU cache policy. 10% of the cache space near the LRU position is allocated as the clean-first region, same configuration as used in [4].
- *ARC*: Adaptive Replacement Cache policy.
- *LRU-WSR*: Least Recently Used-Writes Sequence Reordering cache policy.
- *H-ARC*: Our proposed Hierarchical Adaptive Replacement Cache.


(a) mds_0


(b) wdev_0


(c) web_0


(d) fio zipf


(e) fio pareto

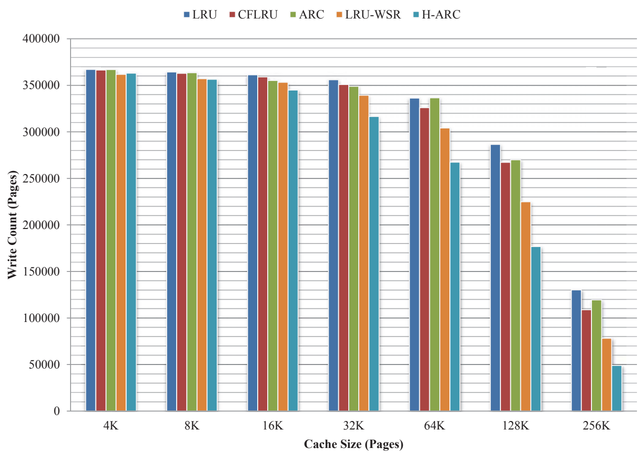
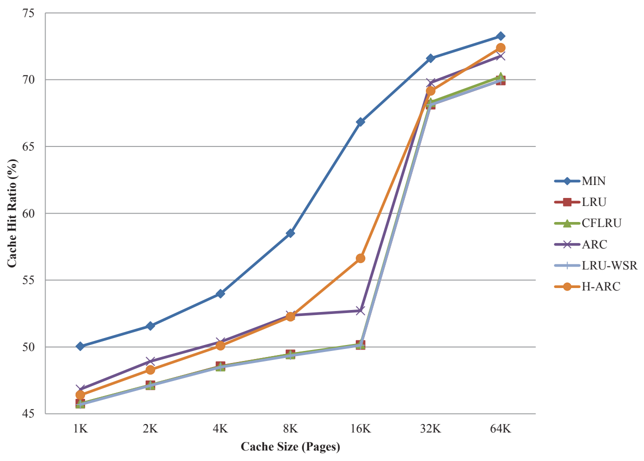

(f) file server

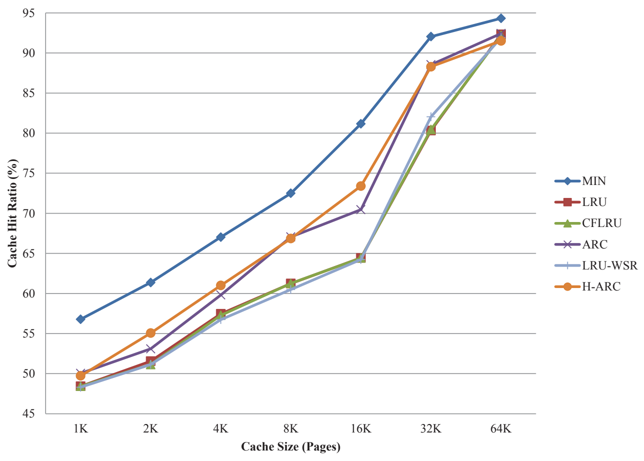
Fig. 2. Storage write count in pages from the main memory to storage devices.

A. Experimental Setup

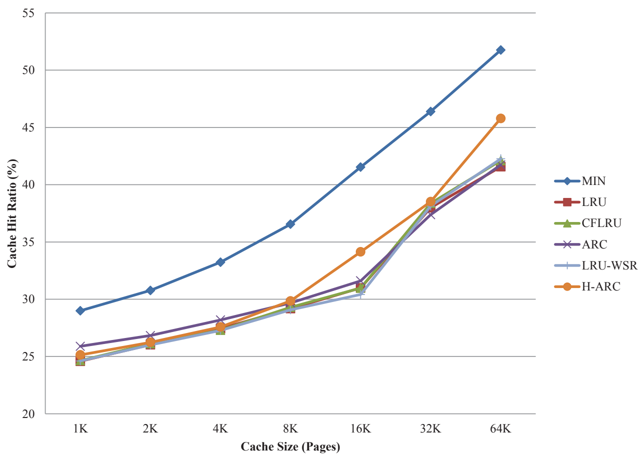
To evaluate H-ARC, we implement it along with the comparison cache policies on Sim-ideal [12] simulator. Sim-ideal accepts a trace file and a config file as inputs. It loads the trace file into an internal data structure (queue) and processes trace



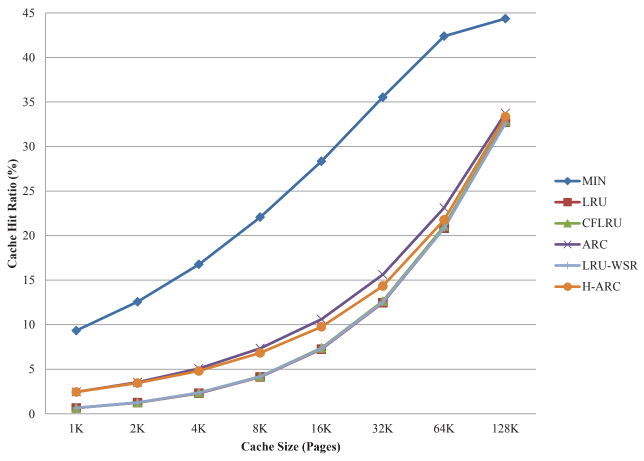
(a) mds_0



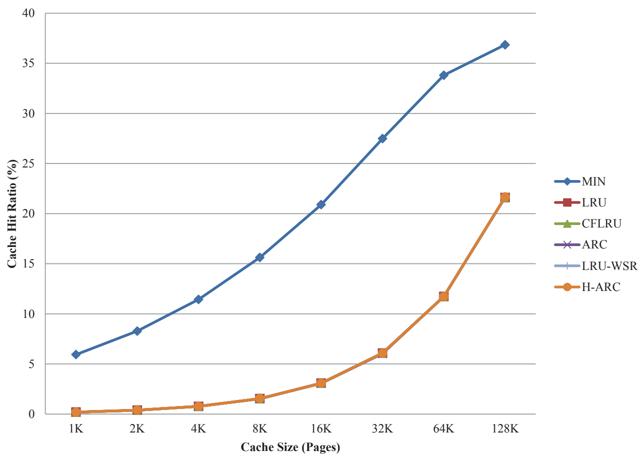
(b) wdev_0



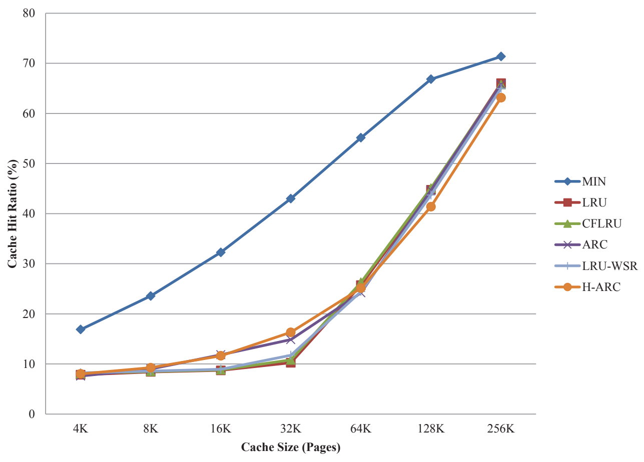
(c) web_0



(d) fio zipf



(e) fio pareto



(f) file server

Fig. 3. Cache hit ratio of both reads and writes.

requests one by one from the queue according to the timestamp of each request. Moreover, the simulator core includes defined cache size, page size, etc. according to a given config file. For the experiments, we configure the size of a memory page to

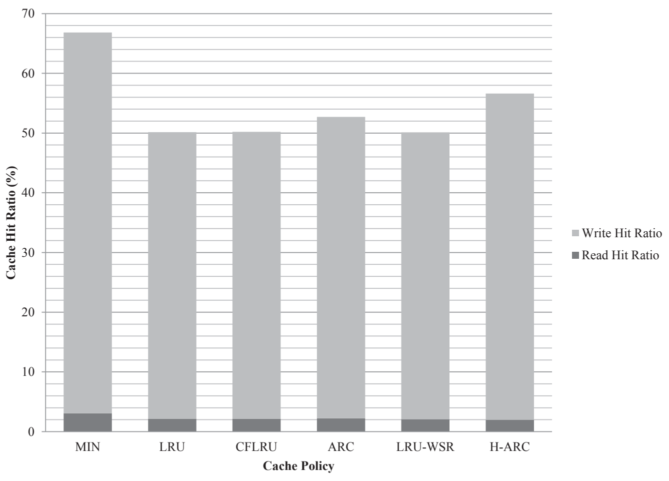


Fig. 4. Cache hit ratio of trace mds_0 under cache size of 16K pages. The read cache hit ratio and the write cache hit are separated.

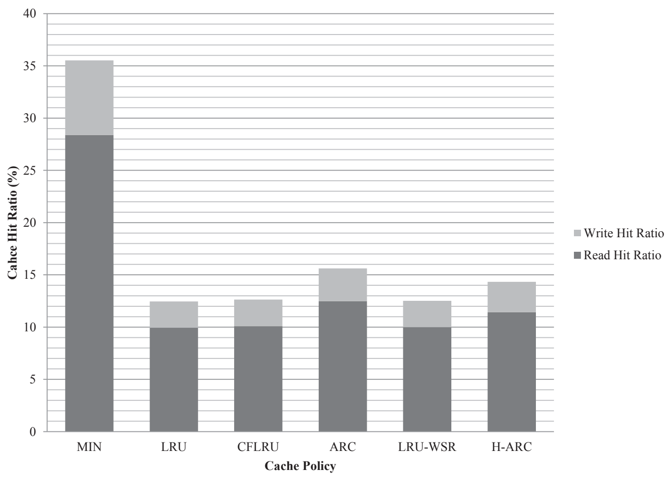


Fig. 5. Cache hit ratio of trace fio zipf under cache size of 32K pages. The read cache hit ratio and the write cache hit are separated.

be 4 KB.

We use two types of traces to evaluate our scheme. The first type is MSR Cambridge traces shared from SNIA [14] and provided by Narayanan *et al.* [15]. MSR Cambridge traces consist of 36 volumes containing 179 disks from 13 Microsoft enterprise servers with different purposes for one week. These traces are classified into 13 categories based on server types. Each category consists of 2 or 3 traces. These traces represent data accesses from a typical enterprise data center. For space efficiency, we show the results of the first volume of traces from 3 categories since the traces in the same category are similar. All the selected traces are write intensive.

The second type is synthetic workload generated by two popular benchmarks: fio [2] and Filebench [3]. Since MSR Cambridge traces are block I/O traces that have been seen by storage, in order to see the effect of traces that are seen by the main memory, we have to generate them ourselves. To achieve this goal, for fio and Filebench, we enable directI/O option. directI/O enables the read/write requests bypass the main memory and go to storage devices directly. Then we collect the traces using Linux blktrace. In this way, even we collect the traces from the storage layer, their actual access pattern is close to accessing the main memory. For fio benchmark, we configure it with 80% read requests and 20% write requests and use two different types of distribution: zipf and pareto. For Filebench, we select a popular model - file server. Table 2 describes these traces in detail.

B. Result Analysis: Storage Write Traffic

Figure 2 shows the storage write count from the main memory (NVM) to storage (SSDs) for different cache policies under various cache sizes. The x axis denotes the cache size, which is the maximum number of real pages the cache can store. The y axis denotes the storage write count, which is in the number of pages written to storage.

We compare five different types of cache policies. Among them, LRU and ARC concentrate on improving cache hit ratio, and CFLRU and LRU-WSR concentrate on decreasing storage write count. H-ARC is our proposed cache policy.

From Figure 2, across six traces, one observation is that for each cache policy, the bigger cache size, the less storage write count. For example, in trace fio pareto as shown in Figure 2(e), the storage write count of H-ARC under cache size of 64K pages is only 26.9% of the amount under cache size of 1K pages. The reason is two fold: (1) with larger cache size, pages can be held in the cache for longer time without the need to be evicted; (2) with larger cache size, better page eviction choices can be made to increase cache hit ratio and decrease the write count.

Another observation is, when the cache size is small, e.g., 1K pages to 8K pages, across the six traces, the differences of storage write count among these cache policies are small. The reason is due to the limited cache space, it's difficult for these cache policies to capture enough information to make better choices. In fact, under small cache sizes, all cache policies perform similarly to the LRU algorithm. Even so, on average H-ARC still performs better than all the other compared cache policies.

On the other hand, when the cache size is big, e.g., 16K pages to 64K pages, the differences of storage write count among these cache policies are quite clear. In fact, H-ARC decreases storage write traffic dramatically. For the extreme cases in traces fio zipf and fio pareto, under cache size of 128K pages, H-ARC has no write count, which means all the dirty pages are kept in the cache and no dirty page is evicted. For the three write intensive traces, which are mds_0, wdev_0 and web_0, under cache size of 16K, on average H-ARC decreases storage write to 73.8%, 74.4%, 80.8% and 76.2% compared with LRU, CFLRU, ARC and LRU-WSR respectively. Under cache size of 32K, on average H-ARC decreases storage write to 68.0%, 71.1%, 82.5% and 82.3% compared with LRU, CFLRU, ARC and LRU-WSR. Under cache size of 64K, on average H-ARC decreases storage write to 53.2%, 57.2%, 56.2% and 69.9% compared with LRU, CFLRU, ARC and LRU-WSR. For the three read intensive traces, which are fio zipf, fio pareto and file server, under cache size of 32K, on average H-ARC decreases storage write to 80.9%, 82.8%, 83.7% and 87.1% compared with LRU, CFLRU, ARC and LRU-WSR. Under cache size of 64K, on average H-ARC decreases storage write to 63.6%, 66.8%, 65.6% and 73.9% compared with LRU, CFLRU, ARC and LRU-WSR. Under cache size of 128K, on average H-ARC decreases storage write by 43.4%, 49.0%, 47.3% and 62.1% compared with LRU, CFLRU, ARC and LRU-WSR respectively.

C. Result Analysis: Cache Hit Ratio

In this paper, the cache hit ratio (in percentage) is the total hit ratio including both reads and writes. The cache hit ratios of these schemes are shown in Figure 3. All the caches are cold started. Note that for a better distinguishable comparison between these schemes, y-axis (hit ratio) of some figures does not start from 0.

MIN is the optimal offline cache policy, which gives the highest cache hit ratio. Among online ones, across the six traces, on average H-ARC and ARC are the two cache policies achieving the highest cache hit ratio. Both of them consider a page's status of frequency and recency, which can detect a hot page versus a cold page better compared with cache policies only consider recency. For LRU, CFLRU and LRU-WSR, their cache hit ratios are almost overlapped, because all of them are based on LRU, which only consider recency.

For some cases, H-ARC has much higher cache hit ratio than ARC which targets maximization of cache hit ratio. To show the reason, we plot a detailed cache hit ratio splitting reads and writes for the case of trace mds_0 under cache size of 16K pages. As shown in Figure 4, the read hit ratio of H-ARC is slightly lower than the read hit ratio of ARC. However, since we favor dirty pages more, and mds_0 is a write intensive trace, the write hit ratio of H-ARC is much higher than the write hit ratio of ARC. Thus, the overall hit ratio of H-ARC is higher than ARC. On the other hand, for read intensive traces (e.g. fio zipf), on average ARC achieves a little higher hit ratio than H-ARC as shown in Figure 5.

V. Conclusion

In this paper, we propose a new cache policy for computer systems using NVM as the main memory and SSDs as storage devices. To explore the merit of non-volatile nature of NVM and extend the lifespan of SSDs, we split the cache into four smaller caches and balance them to their desired size according to a page's status: dirty, clean, frequency, recency. Through various evaluation experiments, the results show our proposed cache policy H-ARC decreases write count significantly and maintains or even increases cache hit ratio compared with previous work.

Acknowledgment

This work was partially supported by NSF Awards: 1217569 and 0934396. This work was also supported by Hewlett-Packard Company and University of Minnesota DTC/ADC Fellowship.

References

[1] Non-volatile dimm. [Online]. Available: http://www.vikingtechnology.com/non-volatile-dimm
[2] fio. http://freecode.com/projects/fio
[3] Filebench. http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench
[4] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha. LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory. In IEEE Transactions on Consumer Electronics, Vol. 54, No. 3, August 2008
[5] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In ISCA '09: Proceedings of the 36th annual International Symposium on Computer Architecture, pages 2-13, New York, NY, USA, 2009.
[6] M. Krause. The Solid State Storage (R-)Evolution. Storage Developer Conference, SNIA, Santa Clara, 2012.
[7] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In FAST '13: Proceedings of the 11th USENIX conference on File and Storage Technologies, San Jose, CA, Feb. 2013.
[8] R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces http://pages.cs.wisc.edu/řemzi/OSTEP/file-disks.pdf
[9] S. Qiu, and A. L. N. Reddy. NVMFS: A Hybrid File System for Improving Random Write in NAND-flash SSD. In MSST '13: Proceedings of 29th IEEE Symposium on Massive Storage Systems and Technologies, Long Beach, CA, USA, 2013.
[10] N. Megiddo, and D. S. Modha. ARC: a Self-tuning, Low Overhead Replacement Cache. In FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, pages 115-130, Berkeley, CA, USA, 2003.
[11] D. P. Bovet, and M. Cesati. Understanding the Linux Kernel. Third Edition. O'Reilly Media, Inc, 2005.
[12] Sim-ideal. git@github.com:arh/sim-ideal.git
[13] DiskSim v.4.0. http://www.pdl.cmu.edu/DiskSim/.
[14] SNIA. http://www.snia.org/
[15] D. Narayanan, A. Donnelly, and A. I. T. Rowstron. Write Offloading: Practical Power Management for Enterprise Storage. In FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies, pages 253-267, San Jose, CA, Feb. 2008.
[16] J. Yang, N. Plasson, G. Gillis, and N. Talagala. HEC: improving endurance of high performance flash-based cache devices. In SYSTOR '13: Proceedings of the 6th International Systems and Storage Conference, ACM, Article 10, 11 pages. New York, NY, USA, 2013.
[17] M. Murugan, and David H.C. Du. Rejuvenator: A Static Wear Leveling Algorithm for NAND Flash Memory with Minimized Overhead. In MSST '11: Proceedings of 27th IEEE Symposium on Massive Storage Systems and Technologies, Denver, Colarado, USA, 2011.
[18] L. Chang and T. Kuo. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems. In RTAS '02: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium. Washington, DC, USA, 2002.
[19] D. Jung, Y. Chae, H. Jo, J. Kim, and J. Lee. A Group-based Wear-leveling Algorithm for Large-capacity Flash Memory Storage Systems, In CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. pp. 160-164, New York, NY, USA, 2007.
[20] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In Proceedings of USENIX 2008 Annual Technical Conference, pages 57-70, 2008.
[21] L. Chang. On Efficient Wear Leveling for Large-scale Flash Memory Storage Systems. In SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing. pp. 1126-1130, New York, NY, USA, 2007.
[22] O. Kwon, and K. Koh, Swap-Aware Garbage Collection for NAND Flash Memory Based Embedded Systems. In CIT '07: Proceedings of the 7th IEEE International Conference on Computer and Information Technology. pp. 787-792, Washington, DC, USA, 2007.

[23] Y. Du, M. Cai, and J. Dong. Adaptive Garbage Collection Mechanism for N-log Block Flash Memory Storage Systems. In ICAT ’06: Proceedings of the 16th International Conference on Artificial Reality and Tel-existence Workshops. Washington, DC, USA, 2006.

[24] J. Kang, H. Jo, J. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In EMSOFT ’06: Proceedings of the 6th ACM & IEEE International conference on Embedded software. pp. 161-170, New York, NY, USA, 2006.

[25] Micron brings NVDIMMs to Enterprise (February 24, 2013). Retrieved September 24, 2013, from http://mandetech.com/2013/02/24/micron-brings-nvdimms-to-enterprise/

[26] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In ISCA ’09: Proceedings of the 36th annual international symposium on Computer architecture, pages 24-33, New York NY, USA, 2009.

[27] W. Zhang, and T. Li. Exploring Phase Change Memory and 3d Die-stacking for Power/thermal Friendly, Fast and Durable Memory Architectures. In PACT ’09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, pages 101-112, Washington, DC, USA, 2009.

[28] H. B. Sohail, B. Vamanan, and T. N. Vijaykumar. MigrantStore: Leveraging Virtual Memory in DRAM-PCM Memory Architecture. ECE Technical Reports, TR-ECE-12-02, Purdue University.

[29] L. Ramos, E. Gorbatov, and R. Bianchini. Page Placement in Hybrid Memory Systems. In ICS ’11: Proceedings of the international conference on Supercomputing, pages 85-95, Seattle WA, USA, 2011.

[30] X. Wu, and A. L. N. Reddy. SCMFS : A File System for Storage Class Memory. In SC ’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1-11, Seattle WA, USA, 2011.

[31] R. Freitas. Storage Class Memory: Technology, Systems and Applications. Nonvolatile Memory Seminar, Hot Chips Conference, August 22, 2010, Memorial Auditorium, Stanford University.

[32] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an Energy-efficient Main Memory Alternative. In ISPASS ’13: Proceedings of 2013 IEEE International Symposium on Performance Analysis of Systems and Software. Austin, TX, USA, 2013.

[33] L. O. Chua. Memristor: The Missing Circuit Element. IEEE Transactions on Circuit Theory, CT-18 (5): 507-519, 1971.

[34] L. A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. IBM Sys. J., vol. 5, no. 2, pp. 78-101, 1966.

[35] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of Optimal Page Replacement. J. ACM, vol. 18, no. 1, pp. 80-93, 1971.

[36] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee. CFLRU: a Replacement Algorithm for Flash Memory. In CASES ’06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pages 234-241, New York, NY, USA, 2006.

[37] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee. FAB: Flash-aware Buffer Management Policy for Portable Media Players. Consumer Electronics, IEEE Transactions on, 52(2):485-493, 2006.

[38] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha. Performance Trade-offs in Using NVRAM Write Buffer for Flash Memory-based Storage Devices. IEEE Transactions on Computers, vol. 58, no. 6, pp. 744-758, 2009.

[39] G. Wu, X. He, and B. Eckart. An Adaptive Write Buffer Management Scheme for Flash-based SSDs. ACM Transactions on Storage (TOS), vol. 8, no. 1, pp. 1:1-1:24, Feb. 2012.