A Regional Popularity-Aware Cache Replacement Algorithm to Improve the Performance and Lifetime of SSD-based Disk Cache

Feng Ye, Jianxi Chen✉, Xuejiao Fang, Jieqiong Li, Dan Feng
Wuhan National Lab for Optoelectronics
School of Computer, Huazhong University of Science and Technology
Wuhan, China
{yefeng, chenjx, fangxj, jieqiong_li, dfeng }@hust.edu.cn

***Abstract*—Flash-based Solid State Drive (SSD) has limitations in terms of cost and lifetime. It is used as a second-level cache between main memory and traditional HDD-based storage widely. Adopting traditional cache algorithms, which are designed primarily depending on temporal locality and popular blocks, to SSD-based second-level disk cache can cause unnecessary cache replacements, which not only degrade the cache performance but also shorten the lifetime of SSD. To overcome this problem, this paper proposes a performance-effective Regional Popularity-Aware Cache replacement algorithm (RPAC). Instead of a single block, the popularity of a region which is constituted by many adjacent disk blocks is recorded and used to determine replacing a block or not. In this way, the spatial locality of disk access is completely leveraged and sequential I/O blocks are gathered in SSD cache. Furthermore, it reduces the number of unnecessary cache replacement and erasure operation on SSD, prolonging its lifetime. We have implemented RPAC in real system and evaluated it by many workloads. Compared to traditional cache algorithms, it improve I/O throughput by up to 53% and reduce cache replacements of SSD up to 98.5%.**

Keywords—SSD; Regional Popularity; Cache Algorithm; Lifetime

I. INTRODUCTION

With the fast improvement of solid state storage technology, Flash-based Solid State Drive (SSD) has been put in a position to diverse storage systems. It has strengths of higher performance, lower energy and lower noise than magnetic hard disk drive (HDD). However, due to its relatively high price and low capacity, SSD is widely used in hybrid storage systems in either extending or caching HDD mode. In caching HDD mode, memory serves as the first-level cache while SSD serves as a second-level cache for HDD, which is also called SSD-based disk cache in this paper.

Traditional cache algorithms take primarily temporal locality and popular data into account to improve hit ratio. It is not suitable to apply these algorithms directly to SSD-based disk cache for two reasons. Firstly, SSD-based cache is a second-level cache, while temporal locality of second-level cache is much poorer than first-level cache. As shown in study work [1] conducted by Zhou et al., accesses to second-level buffer cache are missed accesses from first-level buffer cache and exhibit weaker temporal locality. Munt and Honeyman's file server caching study [3] reaches a similar conclusion about temporal locality in multi-level caches. Hence, frequently used temporal locality based replacement algorithms, such as LRU (Least Recently Used) [2], MRU (Most Recently Used) [4], LRU-k (Least kth-to-last Reference) [5], LFRU (Least Frequently Recently Used) [6], 2Q (Two queue) [7], LIRS (Low Inter-Reference Recency Set) [8], ARC (Adaptive Replacement Cache) [9], will perform poorly on second-level cache. To achieve high performance in lower level caches, all of the above cache algorithms cannot be used directly. Secondly, popular data is mostly identified by the statistics of access frequency per disk block and Frequency-Based Replacement (FBR) algorithm performs better than locality-based replacement algorithms for a back-end disk cache [10], but the statistics of popular blocks need a long period of time for computing access frequency, during which there may be frequent and ineffective replacements among popular and unpopular blocks. This not only degrades performance but also leads to more write operations to SSD, shortening its lifetime.

To address the above problems, we designed a Regional Popularity-Aware Cache replacement algorithm (RPAC) for improving the performance and lifetime of SSD-based disk cache in this paper. Summarily, following contributions are made.

- We have studied several sets of real-world storage traces and characterize their access patterns in different access cycles. Results show that accesses to HDD have poor temporal locality and there are indeed mutual replacements. Meanwhile, most traces have good spatial locality and certain disk regions are accessed frequently in different cycles.
- We proposed a regional popularity-aware cache replacement algorithm for SSD-based disk cache. Unlike prior cache algorithms, cache replacement is determined by the popularity of region a block lies in. RPAC can identify popular regions quickly and adjacent disk blocks are gathered in cache even they were accessed in a long span, thus decreasing the number of unnecessary cache replacements and extending the lifetime of SSD.

978-1-4673-7891-8/15/$31.00 ©2015 IEEE

We implemented RPAC in a real system and evaluated it with *Filebench*. The result shows it has better performance than traditional cache algorithms.

The rest of this paper is organized as follows. Section II analyzes several traces access characteristics. Section III describes the RPAC algorithm in detail. Section IV presents the results of simulation experiments for RPAC by *cachesim* and section V presents the implementation and simulation results based on I/O traces. Related work is discussed in Section VI. Finally, Section VII concludes this paper.

II. TRACES ANALYSIS

To design both performance and lifetime effective SSD-based disk cache replacement algorithm, we need to know the I/O behavior on backend storage primarily. So a few representative I/O traces, including *Mail*, *Webvm*, *Home*, *WebSearch*, *Financial* and *MSNStorage*, were analyzed before designing the SSD-based disk cache replacement algorithm.

Mail, *Webvm* and *Home* are obtained from a project on I/O deduplication [11]. *Mail* is a mail server trace. *Webvm* is webmail proxy and online course management trace. Homes is a random write intensive I/O trace obtained from an NFS server. *Financial* is a random write intensive I/O trace obtained from an OLTP application running at a financial institution [12]. *WebSearch* is a random read intensive I/O trace obtained from a web search engine [12]. *MSNStorage* I/O trace is composed of 4 RAID-10 volumes on an MSN storage back-end file store [13]. Part of data from different traces is selected to analyze characteristics of I/O behavior on backend storage.

To analyze these traces, a statistic cycle is defined by a fixed number of I/O requests firstly. Considering different requests as the statistic cycle (i.e. 500, 2000, 5000 and 10000 I/O accesses), we have analyzed the distribution of requests at the first four cycles on those traces.

By analyzing these traces, the following observations are achieved.

(1) The distribution of I/O accesses to a block is not uniform during different cycles and the average interval time of two accesses to the same block is very long.

Taking *Financial* trace as an example, table Ⅰ shows a part of its popular regions. From the table we can see that one block is accessed in only one statistic cycle and some other adjacent blocks will be accessed in another statistic cycles.

After a profound analysis of popular blocks in one statistic cycle, we also find that popular blocks alternate with seldom accessed blocks or blocks with approximately equal accesses. As shown in Figure 1, particular marks are put on three logical blocks 107829, 206172 and 817875 to illustrate the problem from two aspects. On the one hand, logical block 107829, a popular block, has been accessed 5 times in a period of time in table Ⅰ, while it has only 3 accesses in Figure 1 and another 2 accesses after 2000 accesses. On the other hand, if we consider only three marked blocks, logical block 107829 is accessed alternatively with another two blocks. So, large amounts of invalid replacements may result in no cache hit even though logical block 107829 is popular in the long run. These

TABLE Ⅰ ACCESS FREQUENCIES OF FINANCIAL TRACE AT THE FIRST FOUR CYCLES

LBA	T1	T2	T3	T4
107349	5			
107365	5			
107389	5			
107405	3	2		
107429		5		
107445		3	2	
107469			5	
107477				5
107485				3
107821	2			
107829	5			
107837	5			
107845	5			
107869		5		
107885		5		

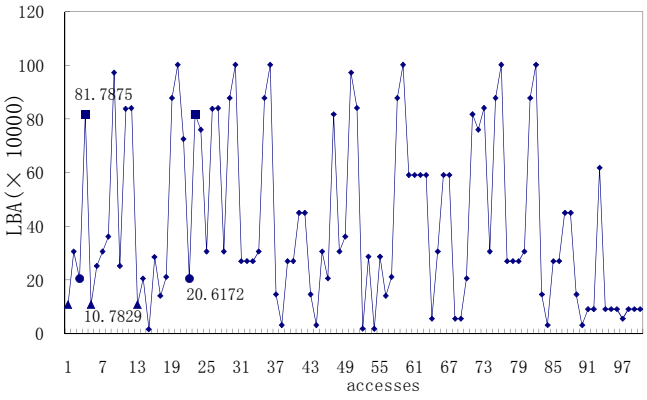
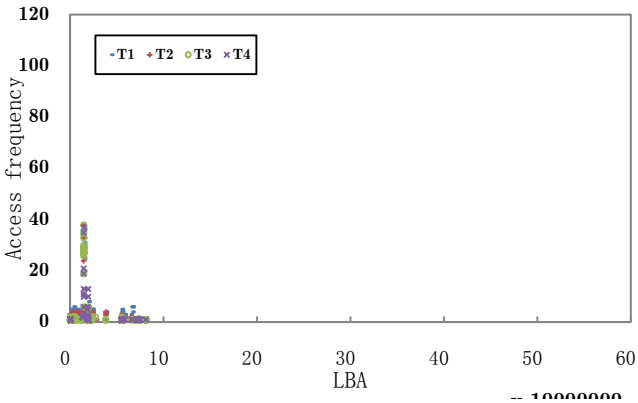


Fig. 1. Access characteristics of hot regions in table Ⅰ for 100 accesses

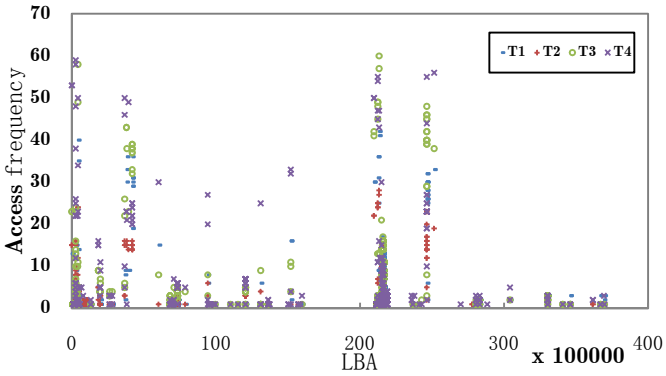
replacements cannot improve performance but increase the number of write operations of SSD instead. That has a non-negligible effect on SSD with limitation of lifetime. If popularity is counted by the granularity of one block, it will take a very long period to accumulate accesses before a predefined threshold of popularity is reached. In the process of accumulation, there may be no cache hit to the block at worst. We compared the analysis results of above-mentioned traces, which share this common characteristic.

As an instance, *flashcache* [14] structures cache space as a set associative hash, where one block will be selected and evicted when the cache set is full and a new block associated to the same set is accessed. In this case, a popular block will be cache hit only one or two times before it is evicted by LRU algorithm. Worse than this, there are a large number of blocks which are accessed only once or twice before evicted. In this case, cache will be replaced frequently and its performance will be degraded seriously.

The distribution of reuse distance of different I/O traces was studied in work [15]. For example, the minimal reuse distance of Home I/O trace is 25, but about 40% blocks' reuse distance is greater than 210 (20% between 210 and 218, 20% between 218 and 225). For read intensive I/O workloads, the average reuse distance of *WebSearch* I/O is greater than 218. This means that the reuse probability of data in SSD-cache may be very low with the limitation of set size.



(a) Mail



(b) Webvm

Fig. 2. Access frequencies at the first four cycles for different traces

(2) Accesses to HDD exhibit a high degree of spatial locality and popular regions remain constant in different statistic cycles.

Fig. 2 shows both scatter plots at different statistic cycles for *Mail* and *Webvm* I/O traces. All accesses to disks are distributed among a few particular regions in these two traces. For *Mail* I/O trace, all accesses are distributed in three regions, where accesses of every statistic cycle are nearly equal; for *Webvm* I/O trace, accesses concentrate on two regions mostly, the beginning and middle segment, and are also approximately equal in these two regions. Other traces we analyzed have the same characteristic.

According to the analyses above, we propose a regional popularity-aware cache replacement algorithm to improve performance and lifetime for SSD based disk cache.

III. REGIONAL POPULARITY-AWARE CACHE REPLACEMENT ALGORITHM

In this section, we introduce the RPAC algorithm in detail, including region division, calculating the popularity of a region and cache replacement strategy.

A. Region Division

In order to make the best of spatial locality and the distribution characteristics of popular blocks, a number of

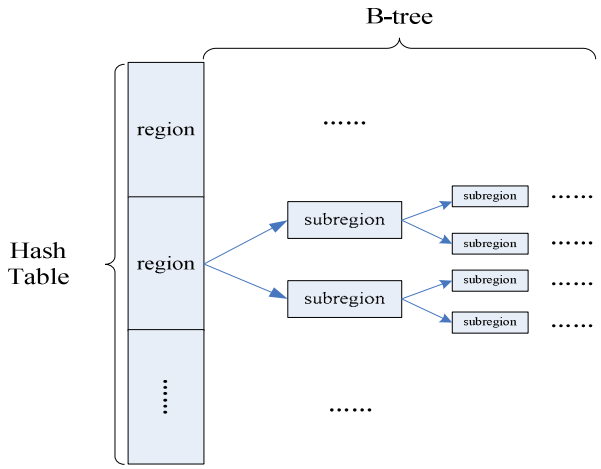


Fig. 3. Data structure to maintain the region-level information

adjacent disk blocks are viewed as a region. Since popular and unpopular data may exist in the same region, a region is

divided into multiple sub-regions to improve the accuracy of popular data identification. Moreover, for the limited capacity of SSD, not all popular blocks in a popular region can be cached and regional popular blocks vary with the time. Thus, a region is subdivided to adapt to the dynamic change of regional popular blocks. As shown in Fig. 3, we use a hybrid data structure, based on Hash table and binary tree, to maintain the region-level information. The size of a sub-region depends on the depth of a binary tree.

B. Popularity Calculation

A Hash table is used to count the number of access on each region rather than each block during each statistical cycle. Before hashing a request, its logical block address is translated into region number.

When a block is accessed, region-level popularity, Pr, is calculated by the following formula:

$$\Pr = \begin{cases} \alpha \times \Pr_0 & (C_A \bmod T = 0) \\ \Pr_0 + f_T & (C_A \bmod T \neq 0) \end{cases} \quad (1)$$

In formula (1), Pr_0 is the region popularity of last cycle and f_T is the access frequency in the current cycle. C_A is the number of blocks accessed. In order to reflect the change of popular data over time, we use a decay factor T. T, a count cycle, represents a fixed number of consecutive requests. After the interval T, popularity values are halved. For the stability of popular regions, accesses in two adjacent cycles are taken into account to calculate the popularity of a region. Considering accesses in current cycle affecting the popularity more significantly, we give a factor (α) for the popularity of the last cycle to accumulate the popularity of a region. This can also accelerate the process of accumulation of the popularity and then reduce cache misses of popular regions caused by the process of accumulation. Based on a large number of previous research and experimentation, the value of a is set 0.5. T, reflecting the speed of hot data update, is determined by different characteristics of workloads.

```
RPAC(Blkio_addr, HashTable)
1  A ← A + 1
2  R1 ← REGION_ADDR_SEGMT(Blkio_addr)
3  Index1 ← HASH1(R1)
4  if (A % T == 0)
5     HashTable(Index1).Te ← a * HashTable(Index1).Te
6     if (HashTable(Index1).Te == 0)
7        REMOVE(HashTable(Index1))
8  HashTable(Index1).Te ← HashTable(Index1).Te + 1
9  Set ← HASH2 (Blkio_addr)
10 S ← SSD_ADDR (Set)
11 if (S == NULL)
12    P ← FIFO_FRONT (Set)
13    n ← FIFO_LENGTH (Set)
14    while (n != 0) do loop
15       R2 ← REGION_ADDR_SEGMT(P)
16       Index2 ← HASH1(R2)
17       if (Index2 < 0)
18          break
19       if (Index1 == Index2)
20          SubRegion1, SubRegion2 ←
                    SUBREGION(HashTable, Index1)
21          if (SubRegion1 != NULL)
22             if (SubRegion1.Te > c* SubRegion2.Te)
23                break
24          else if (HashTable(Index1).Te >
                              c * HashTable(Index2).Te)
25             break
26       P ← P.next
27       n ← n - 1
28    end loop
29    if (n > 0)
30       FIFO_REMOVE (Set, P)
31       FIFO_ENQ (Set, Blkio_addr)
32 else
33    REQUEST_TO_SSD(Blkio_addr)
```

Fig. 4. Process of RPAC algorithm

A. Replacement Strategy

The RPAC algorithm is shown in Fig.4. When a request for block B arrives, the popularity value (Pr_B) is incremented. If cache hit for the request, it is processed normally by accessing from the cache directly. Since cache space is managed by a set associative hash in flashcache, if cache misses and the cache set is full, a block must be evicted. RPAC selects the block from a FIFO list firstly and then compares its regional popularity with block B. Assuming block A has been selected. Rather than comparing regional popularities Pr_B to Pr_A exactly, we compare Pr_B to $c \times Pr_A$, referencing inequality (2). If the regional popularity of block B (Pr_B) is larger than $c \times Pr_A$, then block A is evicted, otherwise, next block in the FIFO list is selected and compared. If none block is satisfied, the request on block B is redirected to the backend storage, i.e., HDD.

$$Pr_B > c \times Pr_A \quad (2)$$

In inequality (2), we choose the value of c larger than 1.0, which will conservatively prevent replacing frequently. In addition, we call two regions with almost the same popularity as "synonymous regions". Replacements occur between synonymous regions cannot improve I/O performance. Instead it incurs more write and erasure on SSD. This is also an important reason to select c larger than 1.0. According to experimental data, RPAC is easy to realize high performance by selecting c from 1.2 to 1.5.

There are two cases during the popularity comparison: comparing popularities of two blocks lying in different regions or comparing popularities of two blocks lying in same region but different sub-regions. If block A and B belong to the same region, RPAC will traverse the corresponding binary the size of cache set. Since RPAC periodically removes the history of accesses in Hash table and binary trees in a round robin manner, it is probable that the popularity value for some block residing in cache cannot be found in Hash table and binary trees. In this case, we assume it is 0.

IV. SIMULATION EXPERIMENTS

In this section, we evaluate RPAC algorithm with *cachesim*, a simulator integrated with RPAC algorithm. We compare RPAC with 5 other algorithms, including traditional cache algorithms (FIFO, LRU, ARC, MQ) and recently proposed LARC algorithm[15]. Finally, the sensitivity of RPAC is analyzed.

A. Workloads

Two kinds of I/O traces (*Mail* and *Webvm*, mentioned in section II) from real systems are used in the simulation. Table Ⅱ gives detailed information of these traces.

B. Results

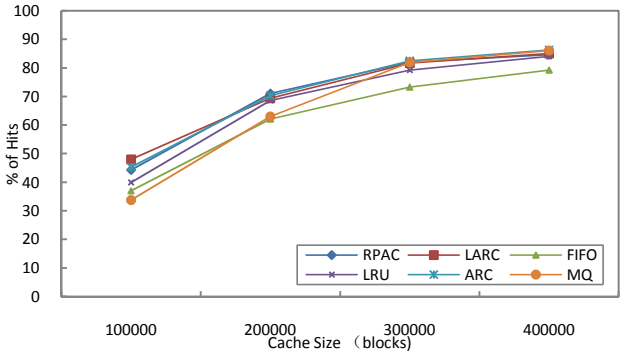
The parameters of RPAC algorithm are set as follows. The region size and the statistic cycle is 4 blocks and 1,600,000 requests for *Webvm* and 8 blocks and 800,000 requests for *Mail*. The other parameters in formula (1) and (2) , α , c and the depth of binary tree, are 0.5, 1.4 and 2 respectively.

Figure 5 and Figure 6 show the hit rates and replacements of different algorithms for *Mail* and *Webvm* respectively. In terms of overall situation, RPAC outperforms all other algorithms significantly.

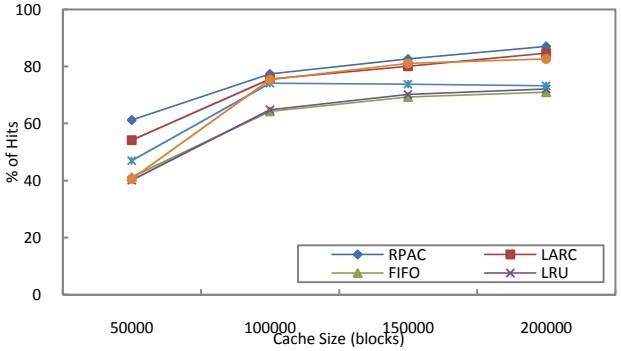
For *Mail*, compared with FIFO, LRU and MQ, RPAC improves hit rate by up to 19.6%, 11.0% and 31.2%. Hit rate of RPAC is almost quite to LARC and ARC. As shown in Figure 6(a), RPAC avoids a large portion of unnecessary replacements, especially when cache size is relatively small. Compared to these algorithms, RPAC reduces cache replacements by up to 66.6%, 92.4%, 92.1%, 91.3% and 92.8%.

TABLE Ⅱ CHARACTERISTICS OF TRACES

	Blocks (×1000)			Requests (×1000)		
	read	write	total	read	write	total
Mail	1,666	685	2,271	2,130	21,492	23,622
Webvm	354	248	549	3,117	11,177	14,294

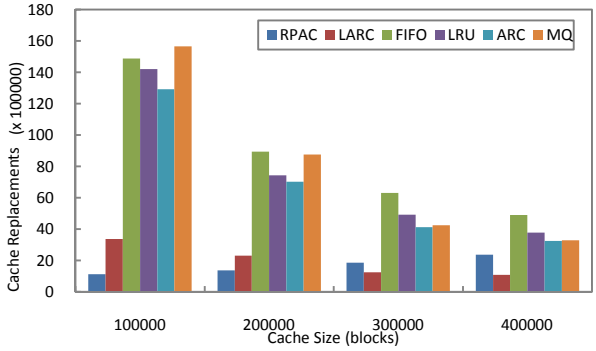


(a) Mail

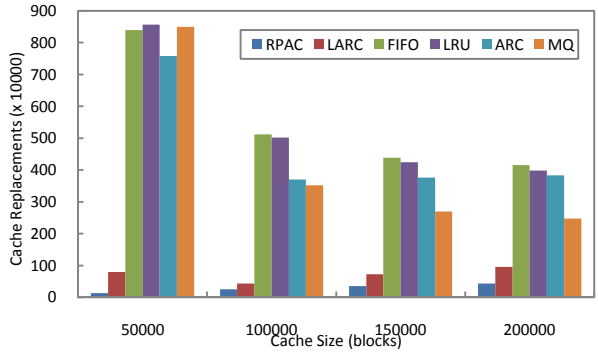


(b) Webvm

Fig. 5. Hit rate of different algorithms under *Mail* and *Webvm* traces



(a) Mail



(b) Webvm

Fig. 6. Cache write traffics under *Mail* and *Webvm* traces

For *Webvm*, RPAC convincingly outperforms all other algorithms. Compared to these algorithms, RPAC improves hit rate by up to 13.0%, 48.2%, 52.7%, 30.3%, 50.9%. Moreover, RPAC reduces cache replacements by up to 84.1%, 98.5%, 98.5%, 98.3%, 98.5%.

With cache size increased, RPAC increases replacements. When cache size is higher than 200,000 blocks, replacements of RPAC are more than LARC for *Mail*. By comparing the analysis, RPAC is more applicable for the situation of small cache.

C. Sensitivity Analysis

Fig. 7 and Fig. 8 show the effect of region and cycle size on RPAC performance. Region size is from 1 to 512 blocks and cycle size is from 100,000 to 1,600,000 requests. RPAC is relatively stable for the range of 2 to 128 blocks. The optimal value of region size for *Mail* and *Webvm* is 8 and 2 blocks respectively. At the point of optimal value, RPAC can not only achieve high hit rate but maintain relatively less replacements. The optimal value, not equal to1, also proves our analysis.

Cache replacements are increased exponentially with the cycle decreased. For *Mail*, RPAC is insensitive to the cycle size between 200,000 and 800,000 requests. For *Webvm*, hit rate is increased with the cycle size. When the cycle size is more than 800000 requests, we do not undertake further testing since the performance is not changed obviously. Therefore, choosing suitable region size and cycle size is important for obtaining high hit rate and less replacements.

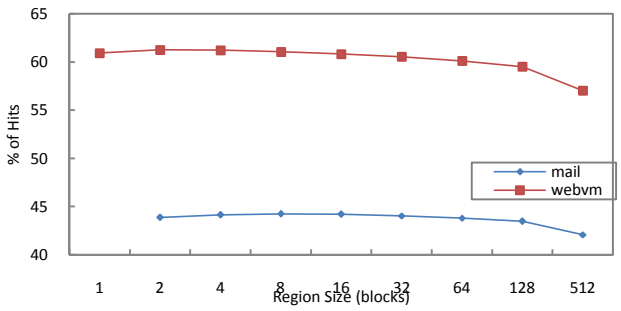
V. IMPLEMENTATION AND EVALUATION

RPAC was implemented on *flashcache*, a Linux kernel module for caching HDD with Flash-based storage device. We evaluated RPAC algorithm by both trace-driven simulating and real system experimenting. The evaluation results are compared with famous cache algorithms widely used in *flashcache* implementations, i.e., LRU, FIFO, and the state of art cache algorithm for *flashcache*, i.e., LARC. Finally, the sensitivity of RPAC is analyzed.

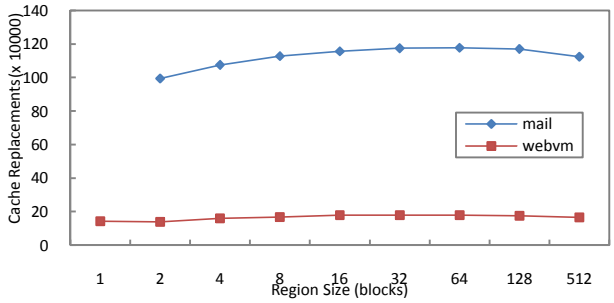
A. Implementation RPAC on Flashcache

Flashcache, an open source project at Facebook, offers a kernel-level solution for providing a block cache for Linux, built as a kernel module on the top of Linux device mapper. It uses faster SSD as cache devices for traditional HDD, RAID or LVM. The cache is structured as a set associative hash, where the cache space is divided into a number of fixed size sets (buckets) and the default size of set is 512 blocks. To find a block, the target hash bucket is probed linearly. The latest released version of *flashcache* supports writethrough, writearound and writeback caching modes and two replacement policies of FIFO and LRU.

We added RPAC into *flashcache* to replace FIFO or LRU. With the supporting of data structure shown in Fig. 3, the popularity of a region is calculated. Based on the regional popularity, the regional popularity-aware cache replacement algorithm is implemented, as shown in Fig. 3. The prototype maintains a hybrid data structure based on hash table and

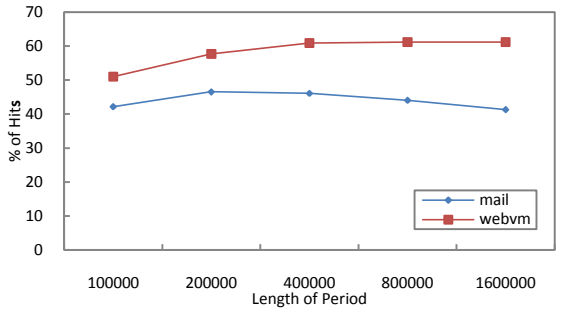


(a) Hit rate

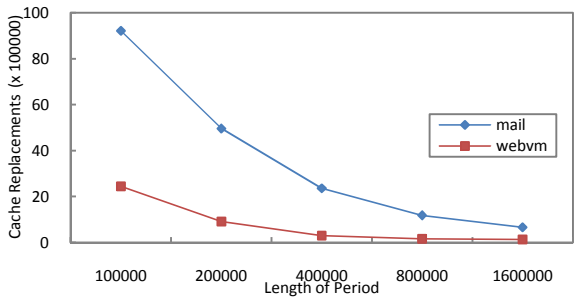


(b) Cache replacements

Fig. 7. Sensitivity analysis of region on RPAC performance



(a) Hit rate



(b) Cache replacements

Fig. 8. Sensitivity analysis of cycle size on RPAC performance

binary tree. Each item in the hash table contains the regional popularity value, a 64bit region number corresponding to each accessed region and two pointers to sub-regions. The popularity values of sub-regions are stored in nodes of the binary tree. It takes only tens of kilobytes of memory.

In our experience, block-level statistics of popularity, compared with region-level statistics, occupies more memory space and take longer time to accumulate out the popular data. Moreover, it cannot capture access information of a sequential block accesses, reducing the possibility of reusing cache data.

B. *Experimental Environment*

Table Ⅲ gives the information of our experimental environment in detail. In order to ensure the difference of deferent storage layers and avoid the effect of memory size, we configure the memory size as 2GB, HDD size as 100GB and SSD cache size varying from 2GB to 6GB. Write-back policy is used in experiments and block size is 4KB, same as the default of *flashcache* released version 2.1.

C. *Evaluation Methodology*

TABLE Ⅲ HARDWARE & SOFTWARE SPECIFICATIONS FOR EVALUATION

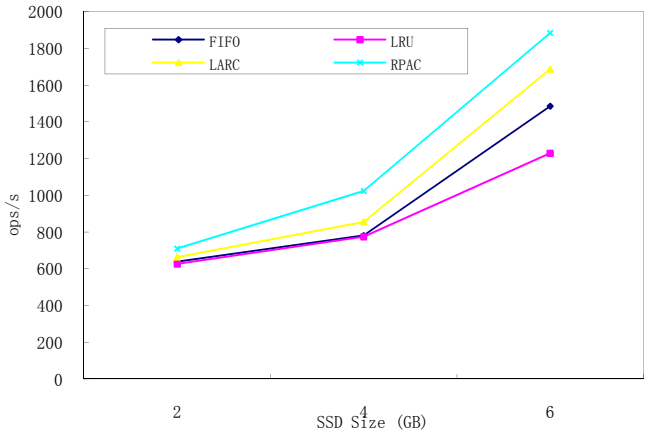
CPU	Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
Memory	Kingston KVR16E11 8GB DDR3-1600MHZ
SSD	Kingston SV300S37A 120GB
HDD	Western Digital WD1003FBYX 1TB
OS	CentOS-6.4-x86_64(Linux Kernel Version 2.6.32)
Implementation Platform	Flashcache -2.1
Benchmark Tool	Filebench -1.4.9.1

Webproxy emulates I/O behaviors of a simple web proxy server. Mixed operations of read, write, append and delete on files in a directory tree are generated by 100 threads. A set of 600,000 files are generated for the evaluation. The average file size is 16KB and the directory width and depth are 1,000,000 and 0.5 respectively. The total size of data is about 9GB.

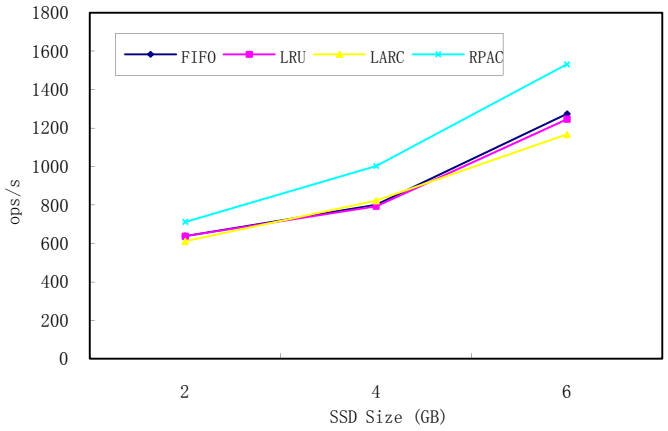
Varmail, similar to *Postmark*, emulates I/O behaviors of a simple mail server that stores each e-mail in a separate file. It generates a set of read, write, delete, create-append-sync and read-append-sync operations in a single directory by multi-threaded mode. Parameters of *Varmail* are configured same as that of *Webproxy*.

To ensure the accuracy of experimental results, some work was done during our evaluation experiments. The HDD was formatted and then *Filebench* was run on it before attaching *flashcache* to system. Thus, the initial environment is fair for every evaluation experiment. Furthermore, Cached data in memory will possibly influence subsequent benchmarks. To eliminate the influence, the system was restarted after each evaluation experiment. Two metrics are used to measure the performance of SSD-based disk cache algorithm. The first one is I/O throughput (operations per second) of the system and the second one is SSD write times per operation ($\frac{SSD\ writes}{total\ operations}$), which determines the lifetime of SSD serving as a cache. Since the number of operations completed varies among benchmark runs, we do not use SSD writes to compare SSD write traffics of different runs directly.

The parameters of RPAC algorithm are set as follows. The region size is 160MB and the statistic cycle is 600 I/O accesses for *Webproxy* workload and 200 I/O accesses for *Varmail*

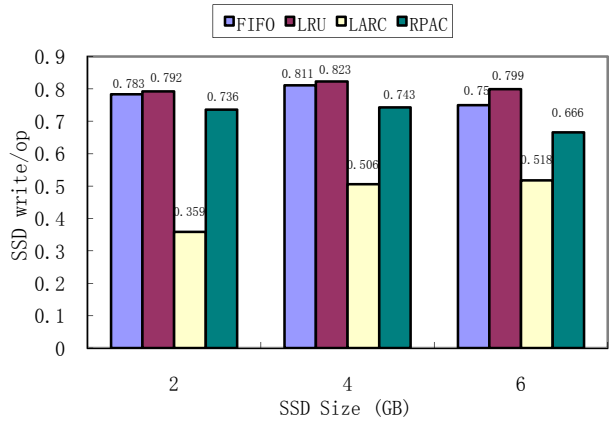


(a) Webproxy

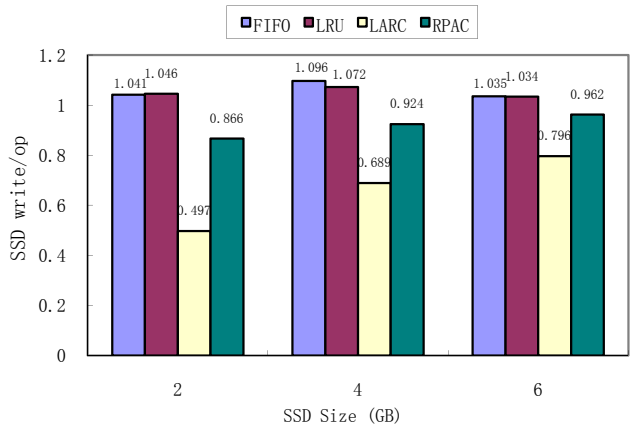


(b) Varmail

Fig. 9. IOPS under *Webproxy* and *Varmail* workloads *Filebench*



(a) Webproxy



(b) Varmail

Fig. 10. SSD writes under *Webproxy* and *Varmail* workloads *Filebench*

workload. The other parameters in formula (1) and (2) , α , c and the depth of binary tree, are 0.5, 1.5 and 2 respectively.

D. Results

We adapt FIFO, LRU, LARC and RPAC algorithms to *flashcache* system and evaluate their I/O throughput respectively. As shown in Fig. 9, RPAC outperforms over all other algorithms under two workloads with different SSD cache size. Compared to FIFO, RPAC improves I/O throughput by 11-31% and 11-25% for *Webproxy* and *Varmail* respectively. It also improves I/O throughput over LRU by 13-53% and 12-26% for two workloads respectively. Especially, RPAC outperforms LARC algorithm, which is a recently proposed algorithm for SSD-based disk cache, by 7-20% and17-31% on I/O throughput for *Webproxy* and *Varmail* workload respectively.

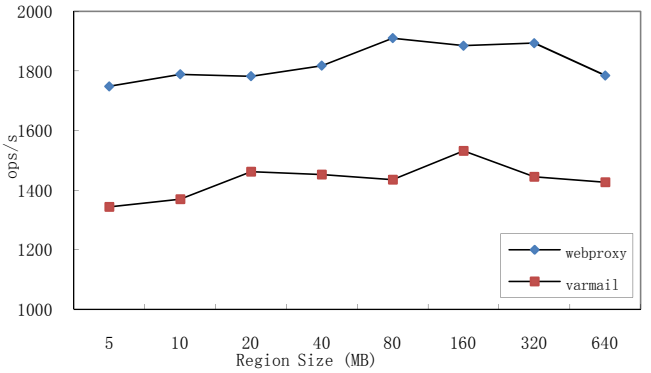
SSD write operations on SSD by different algorithms are compared in Fig. 10. Compared to FIFO and LRU, RPAC significantly reduces the number of SSD writes by up to 17%. It has more SSD writes than LARC but obtains a higher I/O throughput, which is the most important target in some application environments.

E. Sensitivity Analysis

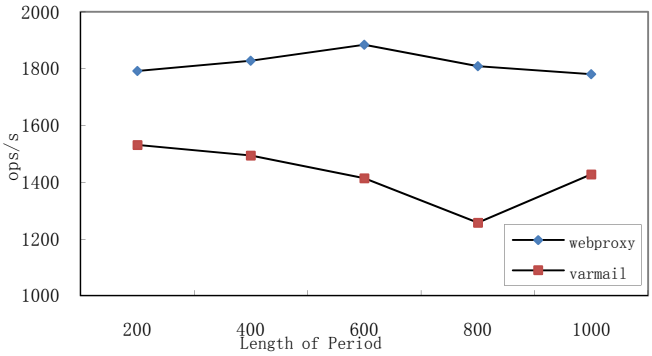
In this subsection, we present how two main parameters of RPAC will affect the performance, which include the region size and statistic cycle. Workloads are configured as that used in the previous subsections.

Fig. 11 (a) shows how I/O throughput is affected by the region size, varying from 5MB to 640MB. When the region size is less than 20MB, the performance begins to decrease obviously. When region size is reduced to 10M, RPAC performs relatively poorly. This also indirectly proves that region-level statistic is superior to block-level statistic. It is likely to result in poor performance if the region size cannot be properly selected. The optimal range of region size for *Webproxy* workload is between 80MB and 320MB, while RPAC shows similar high performance between 20MB and 640MB for *Varmail* workload.

How I/O throughput is affected by varying the statistic cycle of RPAC was also evaluated. Fig. 11 (b) shows our experiment results on statistic cycle varying from 200 to 1000 I/O accesses. For *Webproxy* workload, the best I/O performance was achieved when the statistic cycle is 600 accesses. For the statistic cycle beyond or below 600 accesses, RPAC performance is degraded weakly. For *Varmail*



(a) Effect of region size



(b) Effect of statistic cycle

Fig. 11. Sensitivity analysis of region and cycle size on RPAC performance.

workload, higher performance was achieved while the statistic cycle is between 200 to 400 accesses. As the statistic cycle increases to 600, the performance starts to degrade. Overall, I/O throughput is weakly sensitive to the statistic cycle. Only when the statistic cycle is large enough the popular region and data can be detected out, but much out of date information resides in cache when statistic cycle is too large, which will affect the performance negatively. By our experience, a reasonable statistic cycle is between 200 and 600 accesses. When there are a lot of ephemeral data in and out frequently in some workloads, i.e., cloud workloads, we can use a slightly smaller cycle to accelerate the decay of popularity values. As a conclusion, the choice of the region size and the statistic cycle should be based on different workload characteristics.

VI. RELATED WORK

To combine the advantages of SSD and HDD, various researchers have come up with numerous solutions [16, 17, 18, 19, 20]. By monitoring I/O traffic at runtime, Hystor [21] can automatically identifies critical data having a high priority to keep in SSD, which achieves a significant performance improvement. It selects frequency/request size as the metric of identifying high-cost blocks. Y. Oh et al. studied the effect of over-provisioned space used for garbage collection on the performance of hybrid storage systems [22]. They divided the flash memory cache into three spaces: read cache, write cache and over-provisioned space (OPS). To improve performance of hybrid storage, they design a dynamic scheme to adjust sizes ofthe caching space and OPS. Kim et al. proposed a high-performance hybrid storage system combining SSDs and HDDs, called HybridStore [23].

Cache replacement policies for different workload patterns have also been studied to improve cache hit ratios [24, 25]. SieveStore [26], a new highly-selective architecture, try to reduce allocation-writes with a filter. It achieves higher hit ratios and extends lifetime, compared to unsieved disk-caches. Kim et al. present a buffer management Unified Buffer Management (UBM) scheme that detects sequential and looping references and stores those blocks in separate partitions in the buffer cache [27]. Study work [1] proposed a second-level buffer cache algorithm. There are also many studies on multi-level caching [28, 29, 30]. All of them focus on improving cache hit ratios on DRAM-based cache, which is considered having unlimited erase operation. But for SSD-based cache, write operation will shorten the lifetime of SSD and must be taken attention in cache replacement algorithm.

The concept of region-level statistics for caching is introduced in [31], where Johnson et al. describe the macroblock, groups of adjacent cache block size data. Their main purpose is to improve exploitation of instruction-level parallelism for computing. Different from this, we focus on improving SSD-based disk cache. The other difference is that the former is implemented by hardware, whereas our cache depends on software.

In database systems, Canim et al. [32] compute the popularity of hot regions at the granularity of an extent. Simulations based on DB2 I/O traces, and a prototype implementation within DB2 both show significant speedups. Their work is focused on improving the performance of upper database systems different from *flashcache* in Linux device mapper layer. Our algorithm is not aimed at particular applications and can adopt many kinds of application. The performance is able to be improved as long as the workloads have a high degree of spatial locality. Probably, different workloads get varying degree of performance improvements.

VII. CONCLUSION

This paper proposed a cache algorithm called RPAC for SSD-based disk cache, which can improve the I/O performance and lifetime of SSD simultaneously. A block replacement is determined by regional popularity rather than a single block's popularity.

To validate our algorithm, we implemented it in Facebook's *flashcache* and compared its performance with other cache replacement policies by *Filebench*. Results show that it outperforms traditional FIFO and LRU algorithms on I/O throughput by up to 31% and 53% respectively. Meanwhile, it reduces the number of SSD erase operations by up to 17%. Compared to the state of art LARC algorithm, it also improves I/O throughput by up to 20% and 31% for *Webproxy* and *Varmail* workloads respectively.

The RPAC algorithm is simple in program structure, requiring memory space less than 40KB. Based on our experiments and evaluations, RPAC can be adopted to other workloads with similar characteristics presented in this paper to improve performance of SSD-based disk cache and extend the lifetime of SSD.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their feedback and suggestions for improvement. This work is sponsored in part by National Natural Science Foundation of China (NSFC) under Grant No. 61472153, the National 973 Program of China under Grant No. 2011CB302301, the National 863 Program of China under Grant No. 2015AA015301 and 2015AA016701, the Fundamental Research Funds for the Central Universities, HUST: 2015QN072 and National University's Special Research Fee: 2015XJGH010.

REFERENCES

[1] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. IEEE Transactions on Parallel and Distributed Systems, 15:2004, 2004.

[2] D.L. Willick, D.L. Eager, and R.B. Bunt, "Disk Cache Replacement Policies for Network Fileservers," Proc. 13th Int'l Conf. Distributed Computing Systems, May 1993.

[3] D. Muntz and P. Honeyman, "Multi-Level Caching in Distributed File Systems-or-Your Cache Ain't Nuthin' but Trash," Proc. Usenix Winter 1992 Technical Conf., pp. 305-314, Jan. 1991.

[4] E. Coffman Jr. and P. Denning, Operating Systems Theory. PrenticeHall, 1973.

[5] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," Proc. ACM SIGMOD Int'l Conf. Management of Data, pp. 297-306, May 1993.

[6] D. Lee, J. Choi, J.-H. Kim, S.L. Min, Y. Cho, C.S. Kim, and S.H.Noh, "On the Existence of a Spectrum of Policies That Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies," Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computing Systems, SIGMETRICS Performance Evaluation Rev., vol. 27, no. 1, pp. 134-143, May 1999.

[7] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," Proc. Very Large Databases Conf., pp. 439-450, 1995.

[8] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," Proc. SIGMETRICS, pp. 31-42, 2002.

[9] N. Megiddo and D.S. Modha, "Arc: A Self-Tuning, Low Overhead Replacement Cache," Proc. Second USENIX Conf. File and Storage Technologies, 2003.

[10] J. Robinson and M. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems, 1990.

[11] FIU TRACE REPOSITORY. http://sylab.cs.fiu.edu /projects/iodedup.

[12] UMASS TRACE REPOSITORY. http://traces.cs.umass.edu.

[13] Narayanan, D., Donnelly, A., Thereka, E., Elnikety, S., AND Rowstron, A. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In Proc. of OSDI (2008), pp. 15–28.

[14] Flashcache. [Online]. Available: https://github.com/ facebook/flashcache

[15] S.Huang, Q.Wei, J.Chen, C.Chen, D.Feng, Improving Flash-based Disk Cache with Lazy Adaptive Replacement, in Mass Storage Systems and Technologies (MSST). 2013.

[16] Hong, S., Shin, D. NAND Flash-Based Disk Cache Using SLC/MLC Combined Flash Memory. In Proc. of SNAPI (2010), pp. 21–30.

[17] Kgil, T., Roberts, D., Mude, T. Improving NAND Flash Based Disk Caches. In Proc. of ISCA (2008), pp. 327–338.

[18] Kim, S.-H., Jung, D., Kim, J.-S., Maeng, S. Hetero-Drive: Reshaping the Storage Access Pattern of OLTP Workload Using SSD. In Proc. of IWSSPS (2009), pp. 13–17.

[19] Schindler, J., Shete, S., Smith, K. A. Improving throughput for small disk requests with proximal I/O. In Proc. of FAST (2011).

[20] Q. Yang and J. Ren, "I-CASH: Intelligently coupled array of SSD and HDD," in High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on. IEEE, 2011, pp. 278–289.

[21] Chen, F., Koufaty, D. A., Zhang, X. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In Proc. of ICS (2011), pp. 22–32.

[22] Y. Oh, J. Choi, D. Lee, and S. Noh, "Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems," in Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST12), 2012, pp. 25–25.

[23] Kim, Y., Gupta, A., Urgaonkar, B., Berman, P., Sivasubramaniam, A. HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs. In Proc. of MASCOTS (2011), pp. 227–236.

[24] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In Proc. of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 115–126, 1997.

[25] Y. Smaragdakis, S. Kaplan, and P. Wilson. Eelru: simple and effective adaptive page replacement.SIGMETRICS Perform. Eval. Rev., 27(1):122–133, 1999.

[26] Pritchett, T., Thottethodi, M. SieveStore: A Highly-Selective, Ensemble-level Disk Cache for Cost-Performance. In Proc. of ISCA (2010), pp. 163–174.

[27] Kim, J. M., Choi, J., Kim, J., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References. In Proc. of OSDI (2000).

[28] S. Jiang, K. Davis, and X. Zhang. Coordinated multilevel buffer cache management with consistent access locality quantification. IEEE Trans. Comput., 56(1):95–108, 2007.

[29] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. SIGMETRICS Perform. Eval. Rev., 33(1):145–156, 2005.

[30] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao. Second-tier cache management using write hints. In FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, pages 9–9, Berkeley, CA, USA, 2005. USENIX Association.

[31] T. L. Johnson and W. mei W. Hsu. Run-time adaptive cache hierarchy management via reference analysis. In Proceedings of 24th Annual International Symposium on Computer Architecture, June 1997.

[32] M. Canim, G. Mihaila, B. Bhattacharjee, K. Ross, and C. Lang, "SSD bufferpool extensions for database systems," Proceedings of the VLDB Endowment, vol. 3, no. 1-2, pp. 1435–1446, 2010.