MacroTrend: A Write-Efficient Cache Algorithm for NVM-based Read Cache

Journal:	*Journal of Computer Science and Technology*
Manuscript ID	JCST-1911-10178.R2
Manuscript Type:	Original Article
Keywords:	non-volatile memory (NVM), solid state disk (SSD), cache, endurance
Speciality:	Computer Architecture and Systems

Bao N, Chai YP, Qin X *et al.* MacroTrend: A write-efficient cache algorithm for NVM-based read cache. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(1): 1–30 January 2018. DOI 10.1007/s11390-015-0000-0

MacroTrend: A Write-Efficient Cache Algorithm for NVM-based Read Cache

Ning Bao[1,2], Yun-Peng Chai[1,2*], Member, CCF, Xiao Qin [3] , Senior Member, IEEE, and Chuan-Wen Wang [1,2]

[1] *Key Laboratory of Data Engineering and Knowledge Engineering, Ministry of Education, Beijing 100872, China*
[2] *School of Information, Renmin University of China, Beijing 100872, China*
[3] *Samuel Ginn College of Engineering, Auburn University, Alabama 36830, the United States*

E-mail: baoning@ruc.edu.cn; ypchai@ruc.edu.cn; xqin@auburn.edu; wangchuanwen@ruc.edu.cn

Received November 13, 2019; revised January 25, 2021.

Abstract The future storage systems are expected to contain a wide variety of storage media and layers due to the rapid development of NVM (non-volatile memory) techniques. For NVM-based read caches, many kinds of NVM devices cannot stand frequent data updates due to limited write endurance or high energy consumption of writing. However, traditional cache algorithms have to update cached blocks frequently because it is difficult for them to predict long-term popularity according to such limited information about data blocks, such as only a single value or a queue that reflects frequency or recency. In this paper, we propose a new MacroTrend (macroscopic trend) prediction method to discover long-term hot blocks through block's macro trends illustrated by their access count histograms. And then a new cache replacement algorithm is designed based on the MacroTrend prediction to greatly reduce the write amount while improving the hit ratio. We conducted extensive experiments driven by a series of real-world traces and found that compared with LRU, MacroTrend can reduce the write amounts of NVM cache devices significantly with similar hit ratios, leading to longer NVM lifetime or less energy consumption.

Keywords non-volatile memory (NVM), solid state disk (SSD), cache, endurance

1 Introduction

In recent years, NVM (non-volatile memory) techniques are evolving rapidly. Flash-based SSDs (solid-state drives) [1] have been widely deployed in many enterprise environments. Intel and Mircon have launched their 3D XPoint products[1]. In addition, PCM (phase-change memory) [2], ReRAM (resistive random access memory) [3], and STT-MRAM (spin-torque transfer magnetic random access memory) [4] are on their ways. For the storage media, a trend of diversity has been observed in the storage hierarchy, and future storage systems may contain more layers consisting of DRAM (dynamic random access memory), PCM, ReRAM, STT-MRAM, 3D XPoint, flash, or disks, each of which has different hardware features. As a result, more cache replacement behaviors will happen in a storage system, no matter in the application level, the OS level, or the device level.

However, most of these emerging NVMs have their

Regular Paper

A preliminary version of the paper was published in the Proceedings of DATE 2019.
This work was supported by the National Key Research and Development Program of China under Grant No. 2019YFE0198600 and the National Natural Science Foundation of China under Grant Nos. 61972402, 61972275, and 61732014.

*Corresponding Author

[1] Intel. Intel and micron produce breakthrough memory technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory- technology/.

own problems in writing, especially when performing as cache devices. For example, flash devices, 3D XPoint, PCM, and ReRAM have limited write endurance compared with traditional DRAM caches. Each cell of MLC (multi-layer cell) flash-based SSDs can only be updated for 5,000–10,000 or even fewer times [5] before wearing out for mainstream enterprise SSDs[2] [3] [4]. The write amplification phenomenon of flash chip shortens the lifetime of SSDs further [6]. Intel's latest Optane Memory (i.e., 3D XPoint) just achieves 3 times write endurance of NAND flash[5]. The maximum allowed write count of the ReRAM prototype is 10,000 times less than DRAM, while PCM is 10 millions of times less [7]. What is more, when promoting the storage density of NVMs for practical use, write endurance usually becomes further lower. For instance, the MLC form of PCM cell can only endure 10^5 overwrites [8], 10,000 times lower than the SLC (single-layer cell) PCM prototype. On the other hand, STT-MRAM has another kind of writing problem, i.e., writing cells of STT-MRAM consume too much energy, about 100 times higher compared with DRAM [9].

In this paper, we focus on the situation when NVMs perform as read caches. In this case, NVMs usually suffer the frequent data updating produced by traditional cache algorithms, such as LRU (least recently used), LFU (least frequently used), ARC (adaptive replacement cache), leading to short lifetimes or too much energy consumption for NVMs. Therefore, many write optimized cache algorithms have been proposed, such as SieveStore [10] and LARC (lazy adaptive replacement cache) [11]. However, these algorithms still rely on the same prediction approaches as traditional cache algorithms, such as recency and frequency. For the lack of the information of request access history, the write optimized algorithms reduce the write amount at the cost of unnecessary performance loss.

Motivated by the observation, a new prediction approach aimed to predict the data popularity in a relatively long future is necessary for NVM-based read cache. We propose a new prediction approach named MacroTrend (macroscopic trend). MacroTrend records the access counts of some past time periods of each request to have a macroscopic view of the data popularity trend. Thus, MacroTrend acquires necessary information to predict the long future without much overhead of the storage and computation of the information.

Based on the new long-term prediction method, we propose a new MacroTrend cache replacement algorithm to select and cache long-term hot data. In MacroTrend, infrequent data updating is performed for NVM-based read caches to lengthen the device lifetime or reduce the consumed energy of writing, while the cached blocks keep satisfactory hotness and hit rates between the adjacent distant data updates. Our simulations driven by typical real-system traces reveal that MacroTrend reduces most of the write loads to the NVM cache devices (e.g., 95.3% fewer on average compared with LRU) based on a 24.0% higher cache hit rate on average. Compared with SieveStore, a state-of-the-art cache algorithm with a write reduction, MacroTrend leads to a 50.2% higher hit rate and 85.2% reduction of NVM writes on average.

The contributions of our paper are summarized as follows.

---

[2]Intel solid-state drive DC P3500 series: Product specification. http://www.intel.com/content/www/us/en/solid-state-drives/ssd-dc-p3500-series-product-specification.html.

[3]Micron 9100 PCIe NVM SSD. https://www.micron.com//media/documents/products/product-flyer/9100_ssd_product_brief.pdf.

[4]Samsung enterprise SSD mzplk3t2hcjl (PM1725). http://www.samsung.com/semiconductor/products/flash- storage/enterprise-ssd/MZPLK3T2HCJL?ia=832.

[5]Alcorn P. 3D XPoint SSD pictured, performance and endurance revealed at fms. http://www.tomshardware.com/news/intel-micron-3d-xpoint-memory32434.html.

- We give a deep analysis of cache algorithms and demonstrate the necessarity of a new prediction approach for cache devices with write problems.
- We propose a novel macroscopic prediction approach to give a long-term popularity prediction, named MacroTrend.
- We develop a MacroTrend cache replacement algorithm based on the prediction method. Experiment results show that the algorithm reduces the write amount significantly with little performance loss.

The rest of the paper is organized as follows. In Section 2, we give a deep analysis of cache algorithms and demonstrate our MacroTrend prediction method. Section 3 describes the proposed MacroTrend cache replacement algorithm. Then, the experimental results of MacroTrend and the state-of-the-art cache algorithms are given in Section 4, and Section 5 presents the related work. Finally, Section 6 concludes this paper and discusses our future work.

2 Deep Analysis of Cache Algorithms

NVM-based read caches need cache algorithms with high hit rates and low data updating frequency. However, traditional cache algorithms update data too many times, while the performance of write optimized cache algorithms is likely to be unsatisfactory. To further analyse whether it is unavoidable to reduce the write amount at the cost of performance loss, the insight of the key elements in cache algorithms is required.

The core of cache algorithms can be extracted as two key elements, prediction and updating. Prediction module is responsible for giving prediction of data popularity in the future, and updating module decides which data to be cached and evicted based on the prediction results. As a result, if the prediction module can only predict the data popularity in a short time, only changing the updating behavior leads to bad performance. It is necessary to present a novel data prediction approach containing more information and predicting a long future.

Thus, in this section, we first give a deep analysis of cache algorithms in Subsection 2.1, and further demonstrate how to predict data popularity in the long run in Subsection 2.2.

2.1 Insight of Key Elements in Cache Algorithms

Why do traditional algorithms require so frequent data updating for a read cache? We can analyze this phenomenon from the two key elements that a cache algorithm contains, prediction and updating.

- **Prediction**. A cache algorithm collects necessary accessing information and predicts data popularity in the future.
- **Updating**. According to the prediction result, data updating is performed to replace the predicted coldest data in a cache with the hottest ones each time.

1. Frequent updating + Predicting the short future. Traditional cache algorithms have to choose the frequent data updating manner, because their prediction methods can only effectively predict data hotness in the short future. Although dozens of cache algorithms have been proposed, they always predict data popularity based on recency, frequency, or their mixture. Recency is effective in identifying hot data in the short run according to temporal locality. Frequency is fluctuating all the time. A data block that shows high frequency may be a cooling block with high history accesses, a short-term hot block with dense recent references but few future ones, or a real hot block. It is dif-

ficult to distinguish these data from each other. Thus, the selected data may turn into cold at any moment, making it necessary to update cached data frequently to obtain satisfactory hit rates.

The frequent data updating manner is not a problem for traditional disk cache devices, i.e., DRAMs, which do not have the weakness of limited write endurance. As a consequence, traditional cache algorithms, e.g., LRU, LFU, work well in the past decades.

2. Sparse updating + Predicting the short future. In the new NVM age, however, the limited write endurance or the high write energy consumption raises the requirement for new caching algorithms with high write efficiency to achieve satisfactory hit rates with much reduced writes compared with traditional algorithms. In this case, some new cache algorithms have been proposed to reduce the updating frequency, but they still rely on traditional data prediction approaches. For example, SieveStore [10] and LARC [11] improve the difficulty of entering cache to reduce data updates, whereas they adopt other existing cache algorithms to predict data popularity in the future. Although the data updating behavior is strictly controlled in L2ARC (level 2 adaptive replacement cache)[6], it selects cache blocks according to the traditional ARC algorithm [12]. In WEC (write-efficient caching) [13], the average time when data are kept in a cache is lengthened to cut down the count of cache replacement, but the data prediction manner is still the same as LRU.

Although these algorithms adopt the sparse data updating manner to reduce write amounts of NVM caches, they have negative effects on cache hit rates. The reason lies in that their traditional data prediction methods cannot accurately predict hot data in the long run. For instance, as Fig. 1 shows, a trace analysis based on a real-world trace *websearch* (see more details in Subsection 2.2 and Table 1) shows that LRU and LFU cannot predict the hottest data effectively in a long run.

In Fig. 1, some trace segments containing $100,000$ requests were randomly selected from the trace *websearch* as the access history and LRU and LFU were used to predict the hottest data blocks in the following 100, $1,000$, $10,000$, or $100,000$ requests based on the history information they learned. The Y axes are the overlap percents between the above predictions and the ideal results, i.e., the most accessed data blocks in the selected future periods. The overlap rate measures the accuracy of data prediction. Different lines in the figures indicate different trace segments which were selected randomly.

According to Fig. 1, for LRU, the accuracy of data prediction decreases when predicting longer future in most cases. The accuracy of LFU shows a descending trend or stays at a very low level. When the size of the future window increases, it means that the caching algorithm needs to find the best data blocks in a longer period, which is obviously more difficult. Since both LRU and LFU identify hot data in the short run because frequent cache updating is assumed, it is not weird to see that the prediction accuracy drops when they are utilized to predict the long future. Moreover, LRU and LFU fail to get important information from the long access history. LRU only records the latest access time of each data block and LFU only records the total access time of data. However, predicting the trend of hotness needs much more information. For example, assuming at time t, both block a and block b are accessed and have 100 total accesses, the access density of block a has decreased (block a may be accessed 100 times per ms at first and 1 time per ms at time t) and the access density of block b has increased. The trends of the two

[6]GreggB. L2arc. https://blogs.oracle.com/brendan/entry/test.

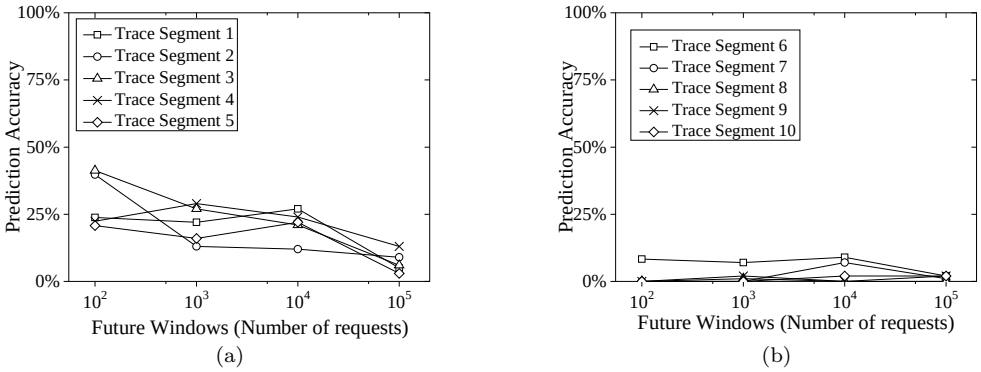

Fig.1. Trace analysis. (a) LRU. (b) LFU.

blocks are opposite but they are considered to be the same for LRU and LFU.

3. Sparse updating + Predicting the long future. The essential reason for the limited prediction accuracy of LRU and LFU lies in that they predict future with too little information which is abstracted from the original accessing records excessively. Illustrated by Fig. 2(a), LRU just keeps a recency order of the latest accessed blocks, while LFU only cares about the total access counts of cached blocks. That is to say, they only have one value for reference when predicting a block's future popularity. Therefore little information may work well in predicting the short future, but it is not enough to predict the long future.

The other extreme is to maintain as much information as possible, e.g., keeping all the request records, as Fig. 2(c) shows. This will bring too much time and space overhead. What is more, too much request information may introduce *noise* that misleads popularity prediction. For example, a hot data block may be misjudged as a cold one in its temporary short inactive duration. A rarely accessed block during a short-lived hot period may be considered as a very hot block. Because it is harder to predict the long future than the short one, we need a method that takes an appropriate abstraction level of access information for the long-term popularity prediction.

2.2 Analysis of Predicting Popularity in the Long Future

To achieve high hit rates with low data updating frequency for NVM-based read caches, a data prediction approach which is able to predict data hotness accurately in the long future is necessary. Thus, in this part, we propose to use an appropriate abstraction called MacroTrend (i.e., histograms of data access counts within several past time periods) for observing data blocks and improving the effect of predicting hot blocks in the long run. MacroTrend contains much richer popularity information than traditional methods, but not introducing too much overhead or popularity noises.

In this part, we first observe the relationship between long-term data popularity and the access count histograms according to a new macroscopic trend perspective in Subsection 2.2.1, then a multi-pattern evaluation method is proposed to select long-term hot data more effectively in Subsection 2.2.2.

The traces used in our trace analysis are listed in Table 1. Among these traces, *websearch* and *fi-*

[7]Umass trace repositor. http://traces.cs.umass.edu/index.php.

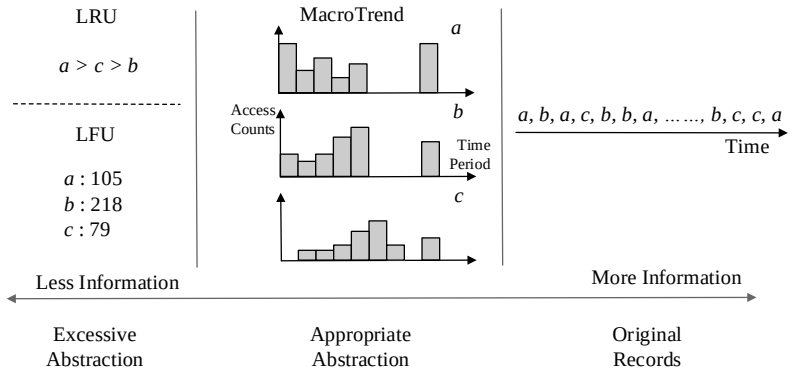

Fig.2. Different abstraction levels of access information in cache algorithms.

nance were collected from a search engine and a financial database[7] respectively and *meta-aggr* and *meta-slct* were collected from the metadata node of Hive by running BigdataBench [14], a benchmark for big data applications, doing aggregation and select operations respectively.

2.2.1 Observing Popularity in Macroscopic Trend

Traditional recency-based cache algorithms (e.g., LRU) observe data popularity in a microscopic perspective (i.e., according to request-level information), which fluctuates frequently and is easy to mislead the judgment on data popularity. Therefore, the algorithms are not appropriate for predicting long-term popularity. Moreover, LRU makes replacement decisions according to a short history window and is unable to utilize the information of a long history. Although the frequency-based algorithms (e.g., LFU) can make the prediction according a long history, they reduce a mass of requests to only one value of a total access count. All the details of the request records are discarded, leading to the confusion of hot blocks and cold ones sometimes.

Therefore, we propose a new data popularity observation perspective called MacroTrend, in which we record a series of periodical access counts to form a data access histogram, shown as Fig. 2(b). The histogram is an access count distribution of some past time periods, containing much more information than a single value of total access count. In this part, we first observe the relationship between the distribution and the long-term data hotness.

Fig. 3 gives an example of analyzing the *websearch* trace in a macroscopic trend view. In this figure, a several-hour request sequence of *websearch* is divided into a list of periods, each of which contains the same number of 10,000 requests. The color of each point in Fig. 3 demonstrates the access count of every block in each period. The darker points mean more accesses in a period. All the blocks are ranked according to the total access counts during the entire service time in a descending order. We sample the blocks evenly in a 6.4% ratio to plot all the results in one figure. In Fig. 3, each column of the dots corresponds to one data block, and the Y axis represents time periods.

Fig. 3 shows that the most-accessed blocks in the left side experience a lot of deep-color periods distributed in the service time. However, the hot data also have some relatively cold stages, such as the history windows marked by the frame $p2$. Moreover, the medium data and the cold data may have hot stages which are close to the hot data. For example, the period $p3$ or even $p5$ may have a similar or even larger total access count compared with $p2$. In this case, it is

Table 1. Four Real-world Traces for our Analysis

Trace name	Application type	Request count	Read ratio
websearch	Search Engine	5,047,596	99.98%
finance	Database	1,555,474	81.82%
meta-aggr	Cloud Storage	144,025	93.41%
meta-slct	Cloud Storage	91,640	94.67%

hard for LFU to distinguish hot, medium, or cold data with each other because it only relies on a single value of the total access count and discards all the details.

Based on some trace analyses, we found that the periodical access count distribution has a strong relationship with the long-term hotness. We first divided user requests in a fixed length (e.g., 10,000 requests as one period), and then a period was considered as an active period when the total access count in this period was larger than a specified threshold, otherwise it belonged to inactive periods. Fig. 4 quantitatively shows the relationship between blocks' active period ratios and their total access counts in the entire service time (i.e., long-term hotness), when the threshold of active periods is set to 3, 5, 10, and 15 times of the average access count of all blocks in one period based on the *websearch* trace respectively. All the four figures with various active-period threshold settings in Fig. 4 show the same trends. Blocks with larger total access counts (i.e., long-term data popularity) usually have higher active period ratios. When the active period threshold is set to 3 or 5 times of the average access count, it is easier to find the most accessed blocks through their active period percentages. This phenomenon confirms that we can discover long-term hot data by observing the access count histogram.

Thus, we propose to observe data popularity in a new macroscopic trend perspective, i.e., separating the request sequence into periods of fixed length and keeping a specified number of past periods' access counts (e.g., 10 periods) for each recently accessed block, as Fig. 2(b) illustrates. Although the basic idea of MacroTrend is to collect more information to predict long-term data popularity, this will not bring much space overhead (see Subsection 3.4 for overhead analysis).

2.2.2 Multi-Pattern Evaluation

So far, we have known the distribution of periodical access counts is helpful when predicting long-term popularity, and the challenge is how to use them for prediction. We can first make an intuitive observation based on the popularity distribution of *websearch* shown in Fig. 3. Typical hot data (see frame $p1$) have large access counts in nearly all periods (i.e., in the dark colors), and typical medium hot (e.g., $p4$) and cold data (e.g., $p6$) have few active periods. These data are easy to identify according to any prediction method, but there are also many cases that are hard to distinguish hot data from others (e.g., $p2$ vs. $p3$ and $p5$).

The hot block indicated by $p2$ has no very hot period in deep gray or black color, but it has uniform user accesses and many active periods. For the medium hot block of $p3$ and the cold block of $p5$, they may have similar total access counts in the whole time window, but their active period distributions are quite different with $p2$, i.e., having fewer active periods in the time window and also having fewer adjacent active periods.

Therefore, we propose a multi-pattern evaluation method in this part to help distinguish hot data from

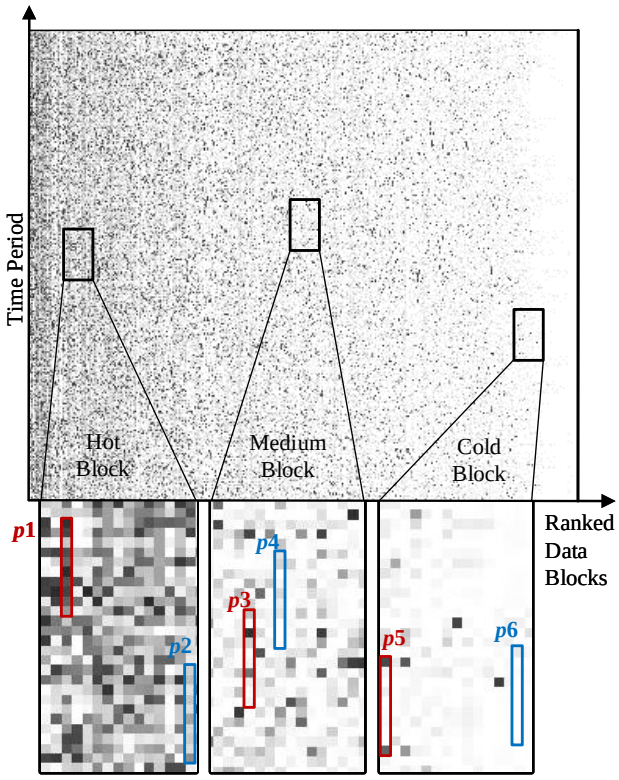

Fig.3. Popularity distribution of sampled data blocks in the unit of 10,000-request period for *websearch*.

others based on the MacroTrend perspective. Long-term hot data usually not only have larger total access counts in a history window, but also have more uniform access distribution, reflected in more active period counts or more adjacent active periods. In other words, a major difference between long-term hot data and other data lies in that the long-term hot data often satisfy more than one feature of active period distribution, whereas the others usually meet none or one feature at most. These features are summarized as LTH (long-term hot) patterns. The more LTH patterns a block satisfies in its recorded past periods, the more likely it is a long-term hot block. The LTH patterns proposed in this paper include the following three patterns.

- Stability pattern (S-pattern). This pattern has a high percentage of active periods in a fixed-length time window of a data block. S-pattern indicates that most of the periods of this block are active periods. Thus, the block constantly being hot is likely to be long-term hot data (see Fig. 5(a) for an example).

- Continuity pattern (C-pattern). As Fig. 5(b) shows, although some long-term hot data do not have as high percent of active periods as S-pattern, they are accessed actively for continuous periods (e.g., 4 periods) in a time window. This is called C-pattern. Note that a medium hot block or a cold block unlikely exhibits this pattern because of its few and scattered active periods. Please refer to Fig. 3 for medium and cold block fragments. Satisfying C-pattern increases the probability of a block being long-term hot data.

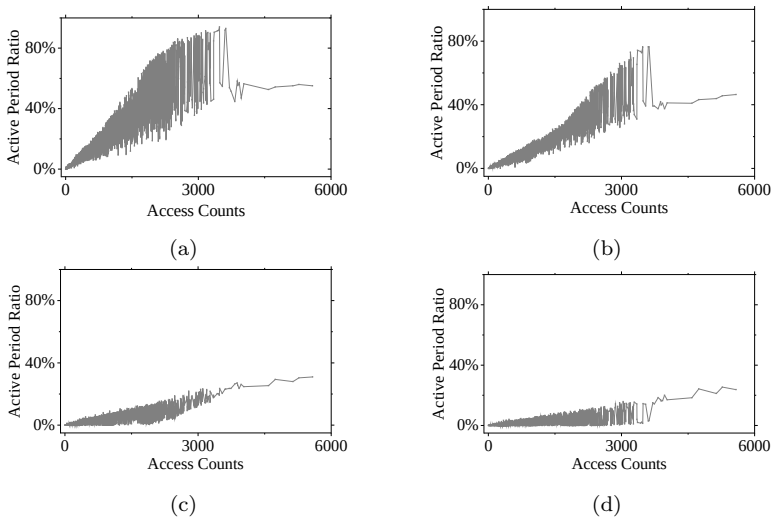

Fig.4. Relationship between blocks' active period ratios and their total access counts of the entire service time. (a) Active Threshold: 3 × Average. (b) Active threshold: 5 × average. (c) Active threshold: 10 × average. (d) Active threshold: 15 × average.

- Irregularity pattern (I-pattern). It means the sum of access counts of all the periods in a past time window is usually very large among all the blocks. A hot block that does not satisfy S-pattern or C-pattern may have an irregular but hot active period distribution (see Fig. 5(c)).

More often than not, the activity distributions of medium hot and cold data do not exhibit the aforementioned LTH patterns. A few examples of non-LTH cases are shown in Figs. 5(d)-5(f).

Only exhibiting one LTH pattern (e.g., S-pattern, C-pattern, or I-pattern) does not mean a data block definitely belongs to long-term hot data, because data with medium activeness (e.g., the frame $p4$ in Fig. 3) may reveal one of these patterns occasionally. Satisfying more than one LTH patterns will greatly increase the chance of a block being a long-term hot block.

Fig. 6 shows the percentage of blocks exhibiting none, one, two, or three of S-pattern, C-pattern, and I-pattern and the blocks are categorized by their access counts for the four real-world traces listed in Table 1. We observe that if a block set is hotter, the percentage of the block following all the three patterns will be larger. This observation indicates that the multi-pattern evaluation method is effective in discovering long-term hot data. The more patterns one block satisfies, the more likely the block belongs to long-term hot data that are worth being cached in NVM caches.

In addition, the framework of MacroTrend prediction method allows incorporating more kinds of LTH patterns in the future, especially the application-specific LTH patterns discovered by some methods like machine learning.

3 The Macroscopic Trend Cache Replacement Algorithm

Powered by the effective MacroTrend prediction method (see Subsection 2.2.2), a new write-efficient cache replacement algorithm also called MacroTrend is proposed in this section to manage NVM-based read caches for longer lifetime or lower write energy consumption. MacroTrend can effectively identify hot blocks in a long run by the multi-pattern evaluation

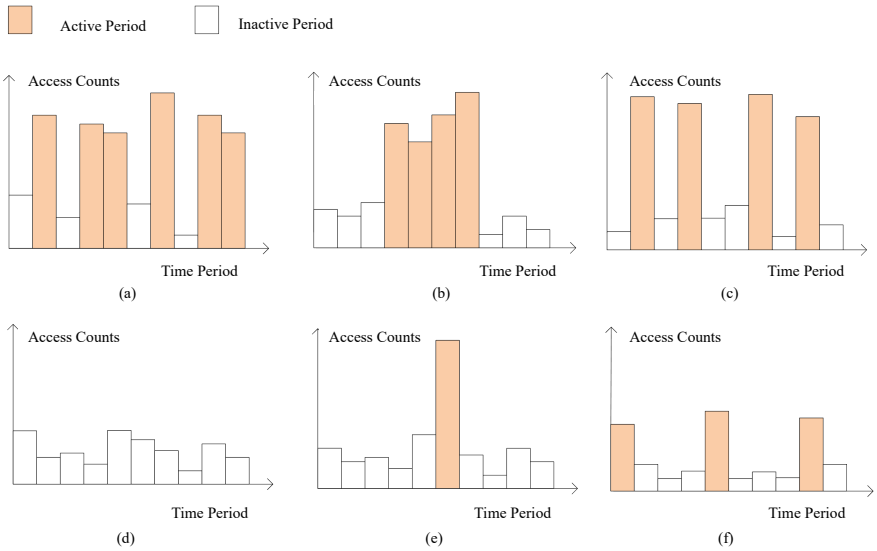

Fig.5. Examples of long-term hot (LTH) patterns and non-LTH patterns. (a) S-pattern. (b) C-pattern. (c) I-pattern. (d) Inactive for all. (e) Few active periods. (f) Not enough sum of access counts.

method, and then keep them in the NVM-based read cache for a relatively long time with lazy data updates.

MacroTrend works in hierarchical storage systems, i.e., managing a flash-based read cache upon a disk storage or an NVM-based read cache upon a flash storage, illustrated as Fig. 7. Because the slow storage layers (i.e., disks or flash-based SSDs) are both block devices, the data in the read cache are also managed, measured, and updated in the granularity of blocks. However, except flash, the other NVMs (e.g., 3D XPoint, PCM, ReRAM, and STT-MRAM) are byte-addressable, and thus the requested data can be fetched from the cache layer in the granularity of bytes, boosting the read behavior compared with flash-based cache.

MacroTrend can be implemented in different kinds of two-tier storage systems, for example, in an application software (e.g., database), in the kernel of operating systems (e.g., the page cache), or in a hybrid device consisting of multiple storage media. In addition, our proposed MacroTrend is a cache replacement algorithm like LRU, but not a complete cache manager.

In this section, the overall architecture and the workflow of MacroTrend are delineated in Subsection 3.1, followed by a series of detailed techniques, including macroscopic observations in Subsection 3.2, multi-pattern evaluation in Subection 3.3, lazy updating in Subsection 3.4. Finally, the implementation details and the overhead analysis of MacroTrend are in Subsection 3.4.

3.1 Architecture

In order to make accurate predictions on long-term hot data, our proposed MacroTrend scheme adopts a macroscopic view on data popularity. First, MacroTrend records the total access count of each block in a long time period. Then, MacroTrend evaluates the blocks based on overall hotness within the large period, avoiding side effects induced by popularity noise in the microscopic level. Second, MacroTrend discovers long-term hot data according to the access count distribution of a series of past periods (i.e., in a coarse-grained way). Evaluations based on the trends of access count changes are much more accurate than those based on a single popularity value at one point. Consequently,

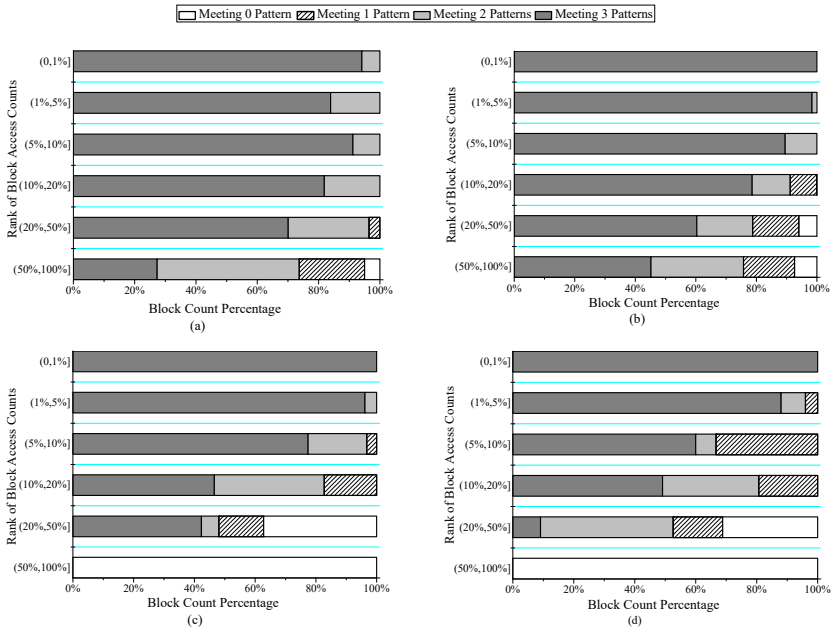

Fig.6. Percentage of blocks that exhibit LTH patterns. (a) *websearch*. (b) *finance*. (c) *meta-aggr*. (d) *meta-slct*.

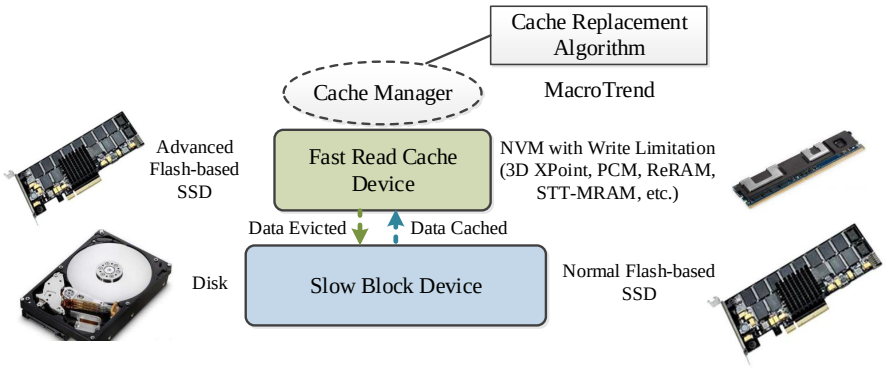

Fig.7. How the MacroTrend cache replacement algorithm is integrated into storage systems.

MacroTrend discovers and keeps long-term hot data in NVM-based read caches, thereby significantly reducing the frequency of updating cached data into NVMs.

Fig. 8 illustrates the architecture of MacroTrend, which is composed of three parts. First, the macroscopic observation module records the access counts of the requested blocks in a coarse granularity. The small squares with different gray levels on the left-hand side of Fig. 8 represent the access count of each period. A dark color means a high access count. Thus, we obtain the popularity distributions of accessed blocks to identify the long-term hot data. Note that no matter a request is hit by the NVM cache or not, it will always be recorded.

Second, the multi-pattern evaluation module matches the access-count distribution in the past periods of each recorded block with some given good patterns of long-term hot data (see, for example, S-pattern, C-pattern, and I-pattern in Subsection 2.2.2). According to the count of the matched LTH patterns, the multi-pattern evaluation module places the blocks into different positions of an NQ (NVM queue) for the

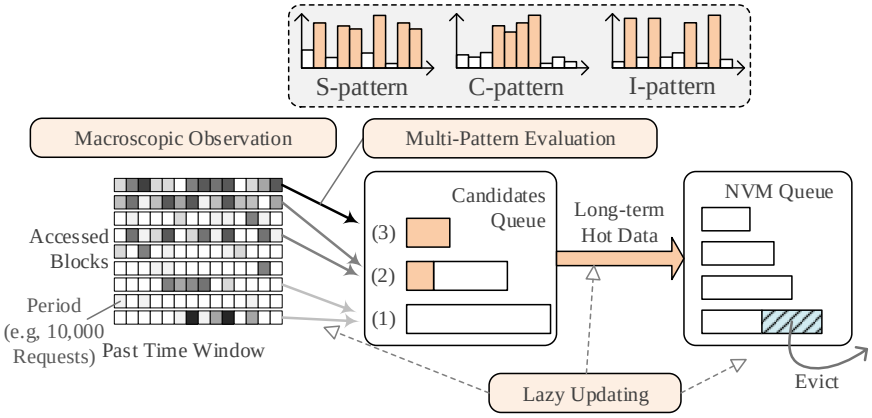

Fig.8. Architecture of MacroTrend.

already cached blocks in NVMs or a CQ (candidate queue), which buffers potential long-term hot data for cache replacement. NQ and CQ both maintain three subqueues, because blocks may satisfy three, two, or one LTH pattern, respectively. The more patterns a block matches, the higher priority the block has for staying in or entering NVM caches. Blocks in a subqueue of NQ or CQ are sorted according to the recency of the period when its latest access locates in. Note that CQ and NQ both belong to temporal data structure for the current round of cache replacement, and will be released after finishing the data updating.

Finally, the lazy updating module changes the cached data updating mode from the request level to the period level, at which there are tens of thousands or more requests, reducing the NVM writes significantly. The lazy updating interval is set in the same level as periods, and there is a limit quota of updating cached blocks (denoted as lazy updating throttling) for each round.

The workflow of MacroTrend is as follows. Once a request arrives, the macroscopic observation module updates the access count of the current period. Each time when a lazy data update is triggered, we generate the latest version of CQ and NQ for candidates and already cached data, according to the method of multi-pattern evaluation, respectively. And then, NQ evicts a specific number (i.e., lazy updating throttling) of the coldest data blocks from NVMs and fetches blocks with a high priority in CQ.

3.2 Lightweight Macroscopic Observation

The lightweight macroscopic observation module contains a record of the access counts of ten history periods and one current period for each accessed block to illustrate the long-term popularity distribution of this block. The data structure is implemented as a hash table in RAM (random access memory) and updated per request.

This module is designed to form a macroscopic popularity distribution by recording the total access counts of each period. Let us use a simple example to describe its procedure. We assume that there are five requests in each period, and the request sequence is a, b, c, a, d; b, a, a, b, a; c, b, e, e, d. The popularity distribution of block a in the three periods is $2, 3, 0$, and that of block b is $1, 2, 1$. Unlike the traditional caching schemes (e.g., LFU) that only record a single value of the total access count to evaluate popularity, MacroTrend decides if a block is likely to be a long-term hot block by comparing the block's popularity distribution of the past periods with some given LTH patterns (see Subsection 3.3).

Fig. 9 shows that the past time window of one block usually contains ten history periods and one current period. When a block is accessed, the access count of the current period increases by one. If the current period reaches its end, this period becomes a history period and the oldest history period (e.g., period 1 in Fig. 9) is removed from the past time window. Meanwhile, a new current period is created to record new access counts.

The period length of MacroTrend is usually set as tens of thousands of user requests. The setting depends on active data size of a storage system. If there are a large number of unique accessed blocks, the period should be set to a large value. Otherwise, the access count in each period is too small to be recognized according to the long-term hot data patterns. Although the optimal setting of the period length is difficult to obtain for a given application, we have found that there are some empirical values of the period length that can achieve satisfactory effects in most cases (see Subsection 3.5.2 for more). In order to reduce the space cost of recording access information, only the blocks that have been accessed during the current period are required to be tracked rather than logging all the blocks in storage devices.

3.3 Multi-Pattern Evaluation

Recalling Subsection 2.2.2, there are three LTH patterns adopted in this paper, namely S-Pattern, C-Pattern and I-Pattern, for discovering long-term hot data. MacroTrend uses several thresholds to decide whether a block satisfies these patterns. These thresholds are self-adaptive and the implement details can be found in Subsection 3.5.

Fig. 9 shows that each time when data updating of an NVM cache is triggered, MacroTrend scans all the recorded information in the history time windows to calculate the following metrics to decide the number of LTH patterns that the accessed blocks satisfy.

S-Pattern. The percentage of the active periods in a time window is compared with a specified threshold of active period percentage of S-pattern. If the percentage of the active periods is no less than the S-pattern threshold (e.g., 50%), the block will satisfy S-pattern.

C-Pattern. If a block has at least a specific number of continuous active periods (e.g., 4) in the time window, the block will satisfy C-Pattern. After obtaining the active period distribution, we compute the value of the maximum continuous active period counts in the observing window to check if a block satisfies C-Pattern.

I-Pattern. A block is considered to be a long-term hot data candidate with an irregular pattern, if the block's total access count in the past time window is larger than a specified threshold, no matter how its periodical access counts distribute. Note that when a block only has access records since the 5-th history period, the access counts of the previous periods are set to 0 representing that there is no access to the block.

After evaluating the LTH pattern count that blocks satisfy, MacroTrend discards blocks that do not match any pattern. The other blocks are placed into the different subqueues of CQ or NQ according to the number of satisfied patterns. Of course, the more LTH patterns a block matches, the more likely the block is to be kept in or fetched into NVM caches.

Note that the access count threshold of active periods is self-adaptive in MacroTrend. It is determined by the actual block accessing status (see Subsection 3.5.2 for more details).

3.4 Lazy Updating

In MacroTrend, we perform a lazy data updating manner, i.e., a certain amount of cached blocks will be replaced with new ones every one or multiple periods,

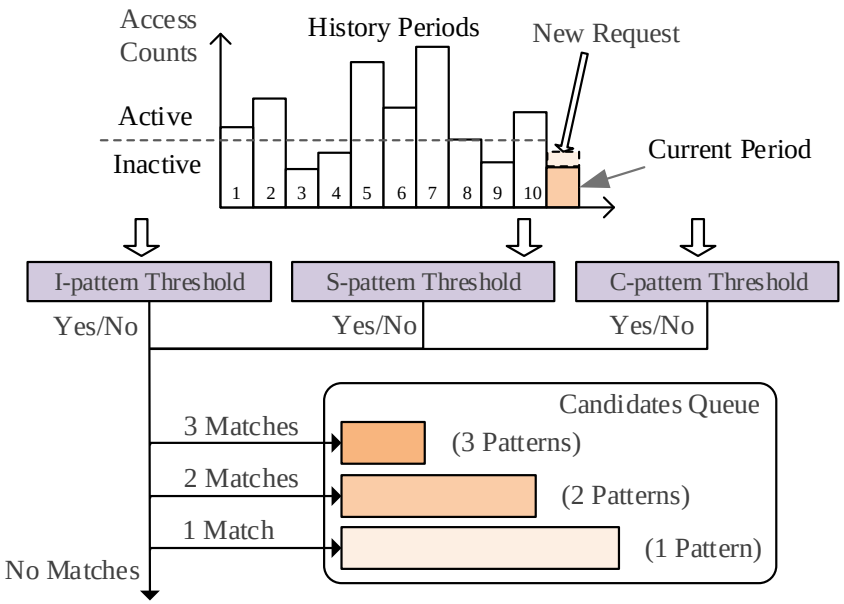

Fig.9. Workflow of the macroscopic observation module and the multi-pattern evaluation module.

which usually contain at least tens of thousands of requests because most cached data in NVMs can keep hot for a long time. In our implement, The cache data updates are triggered per period.

When a data updating process is triggered, the best blocks of CQ are fetched into NVM, resulting in evicting the worst data blocks in NQ with the same amount. The best blocks are the blocks satisfying the most patterns in CQ, and thus the blocks of the subqueue matching 3 patterns of CQ have the highest priority to be fetched into the NVM cache. The maximum number of the updated blocks each time is limited by a value of lazy updating throttling, which is used to avoid an excessive number of unnecessary data updates. The impacts of different throttling settings can be found in Subsection 4.6.2.

3.5 Implementation Details and Overhead Analysis

3.5.1 Data Structure and Workflow of MacroTrend

The most important data structure of MacroTrend is history table, i.e., a hash table to record blocks' access counts in history. It is implemented as a hash table, whose key is the data identifier and the value is a list of 11 access counts. 10 of them are the access counts of the 10 past history periods, and 1 is for current period whose requests have not reach the period length. Another hash table is used for looking up the cached data blocks, similar to other cache algorithms.

And there are some temporal data structures that only exist in the process of cache updating, including the above mentioned *CQ* and *NQ*. Both *NQ* and *CQ* usually contain three subqueues for the blocks that satisfy 3, 2, or 1 LTH patterns respectively.

The detailed workflow of MacroTrend can be divided into two major parts, record operations and update operations. When a request arrives, MacroTrend performs record operations to record necessary information with low overhead. Update operations are processed when there is periodical data updating for the NVM cache. In other words, record operations are conducted at the request level and update operations are conducted at the period level.

Record Operations. When a request arrives,

MacroTrend first looks up the hash table of cached blocks. For a write hit, the hit block in the cache becomes out-of-data and will be discarded, because MacroTrend only manages NVM-based read caches, assuming write requests are served by other modules. For a read request, we will increase the access count of the current period in the history table for the target block. The following processes are very similar to other cache algorithms.

When the accumulated read requests of the current period have reached the specified period length (e.g., 100,000), the current period becomes a history period, and the record of the oldest history period will be reset and used as a new current period.

Update Operations. When MacroTrend starts an updating procedure, it needs to decide which data to be updated into cache and which data to be evicted from cache. We first create two temporal structures, i.e., NQ and CQ, and then scan the history table to calculate the count of satisfied LTH patterns of all the recorded blocks. The already cached blocks will be put into NQ, while the others will be put into CQ.

Assuming MacroTrend's lazy updating throttling is N, the top N data blocks in CQ form the updating data set, in order to replace the worst N blocks in NQ. During this period, the values of all the self-adaptive parameters are also calculated and recorded for the use of the next period. The detailed calculation will be introduced in Subsection 3.5.2.

There are two reasons to use the value in the next round of data updating instead of the current one. First, calculating the values requires parsing the whole history table, which has a large time overhead. This will put off getting the plan of data updating and the procedure of cache replacement. Second, the deviation of the temporal locality of workloads tends not to change so greatly. During our experiments, we find that the values of the parameters are similar between consecutive periods.

Although the above update operations are complex, they happen in a low frequency and can be processed in background.

3.5.2 Parameter Tuning

This part introduces how to tune the parameters of MacroTrend. There are five important parameters in MacroTrend, 1) the period length, 2) the threshold of active periods, 3) three thresholds of three LTH patterns.

In our implementation, some cached blocks are updated after each period is ended, thus the period length influences both the update frequency and the data access histograms used for identifying long-term hot data. When the period length is too large, more memory space and time are needed for recording and computing of the history information, and the data updating frequency may be too low. If the period length is too small, the history information may be not enough for long-term hot data prediction.

In fact, the best period lengths vary with different applications, and it is hard to get the optimal value of each given application. However, there are some empirical values of the period length that can provide good enough effects for most traces. According to our experiments, 100,000 is an appropriate length for most cases. More details can be found in Subsection 4.6.

The other four parameters of MacroTrend are all self-adaptive. The threshold of active periods is a value determined by the access distribution of each trace. Each time when a round of cache updating is triggered, assuming N is the count of updated cache blocks each time (i.e., the value of lazy update throttling), we will order the access counts of the history periods and the N-th access count will be determined as the active pe-

riod threshold of this period. The setting of the other three patterns is similar. The thresholds of S-pattern, C-pattern, and I-pattern are all set as twice of the average values of the history periods.

3.5.3 Space Overhead

MacroTrend has a low space requirement for the metadata maintenance, including the history table, the hash table for the cached blocks, NQ and CQ. For example, assuming each block is 16 KB (kilobyte), an 8-byte identifier can be used to pinpoint any block of a storage system with 16 PB (petabyte) active data. Assuming that there are 100,000 requests in a period, the past time window maintains 10 history periods, and there are 60,000 unique accessed blocks in one period on average, the space overhead of history table, NQ, and CQ is analyzed as follows.

- The history table only needs to record the blocks that have been accessed during the past ten periods and the current period, namely, 660,000 unique blocks. The access count of a block in one period needs 2 bytes to store, thus the past time window only requires 13.2 MB (megabyte) (660,000 $\times$ (8+2) $\times$ 2) space in RAM (assuming the hash table requires two times larger space than the stored items).
- The hash table of the cached data has similar space overhead compared with other cache algorithms.
- CQ stores the candidate blocks which are a partition of the blocks recorded in the history table. And NQ stores the metadata of cached blocks. As a result, the space overhead of NQ and CQ is limited and MacroTrend keeps the memory for CQ and NQ to avoid the frequent memory allocation operations.

Thus, the additional space overhead introduced by MacroTrend is small.

3.5.4 Time Overhead

The time consumption of MacroTrend can be divided into two parts.

1) MacroTrend's additional runtime to process each request lies in adding a block's access count in history table. The time cost is $O(1)$ for both searching and inserting into the hash table. As a result, compared with traditional algorithms, MacroTrend brings little additional time overhead.

2) Occasional process time (e.g., once for hundreds of thousands of requests) for each round of MacroTrend's lazy update comes from organizing CQ and NQ and updating data blocks in NVM cache. To manage CQ and NQ, MacroTrend needs to traverse the history table to decide blocks' locations in CQ or NQ. The time complexity is $O(M \times logM)$, where M is the average capacity of the history table.

Importantly, most of the updating time of NVM cache lies in the I/O time. Thus, MacroTrend can reduce the updating time significantly since it has achieved much fewer cache updates compared with other cache algorithms. What is more, all the above operations can be run in background, without obvious impacts on the system performance.

4 Evaluation

In this section, we will demonstrate the evaluations on our proposed MacroTrend algorithm and a series of existing cache algorithms. For the page limit of Figs. 11–19, MacroTrend is shortened as MT in this section.

After introducing the experimental setup in Subsection 4.1, the overall results under typical configurations are given in Subsection 4.2 followed with the results under different NVM cache size settings in Subsection 4.3.

Subsection 4.4 explains why MT can achieve a good effect through a deep quality analysis of cached data and Subsection 4.5 demonstrates the average response time of NVM devices. Finally, the impacts of several parameters of MT are revealed in Subsection 4.6.

Since MT is a general write-efficient cache algorithm for a variety of NVM devices, including flash NAND, PCM, ReRAM, and STT-MRAM, the simulator does not include the device-specific features.

4.1 Experimental Setup

We conduct extensive experiments using a trace-driven simulated caching system. The simulation platform is designed and developed by us, simulating the behaviors of a cache system including request processing, metadata maintenance, cache replacement, statistics, and etc.

The system is developed with about 3,000 LOC in Python.

Besides MT, LRU, LFU, ARC, pdLRU (PeriodLRU), LARC and SieveStore are implemented in our system. LRU is the most widely utilized cache algorithm for its simplicity and good performance.

LFU is a good representative of the traditional caching replacement algorithms. ARC [12] is optimized for NVM devices which combines LRU and LFU to adapt to different workloads. As a common-sense cache algorithm, pdLRU employs LRU for data popularity prediction and updates the same amount of data as MT. Among the advanced write-optimized caching schemes, LARC and SieveStore are good representatives of the data-filtering cache algorithms, while MT also fits into the same category.

In this part, we avoid using the traces in Table 1 which were already used in our previous analysis in Subsection 2.2 to motivate the design of MT. Instead, MSR (Microsoft Research) Cambridge traces [15] collected from typical enterprise data centers are replayed in our experiments. As Table 2 shows, our evaluations cover all the 12 application categories of the MSR-Cambridge traces .

The comparison between MT and these schemes can reveal the importance of identifying long-term hot data for achieving high hit rates with limited data updating. Thus, two performance metrics are employed to evaluate these cache algorithms, the cache hit ratios and the write amounts of cache devices. The write amount directly affects the NVM lifetime or the write energy consumption of an NVM device like STT-MRAM.

4.2 Overall Performance

In the first group of experiments, we present the overall performance of MT along with other algorithms.

Figs. 10 and 11 illustrate the hit rates and the NVM write amounts of read caches of the 12 real-world traces listed in Table 2 and the average values of them respectively. The cache size is set to a default value, i.e., 10% of the storage capacity. The period length and other parameters of MT are set based on experience. And for a fair comparison, MT and SieveStore make decisions according to the same amount of history data access information. In addtion, the miss count thresholds of SieveStore for data filtering are adjusted to achieve similar cache hit rates compared with LRU and MT.

As Fig. 10 plots, MT achieves either the highest (e.g., *rsrch*, *stg*, *wdev*, and *web*) or very close to the highest (e.g., *ts*, *mds*, and *avg*) cache hit rates among all the evaluated algorithms. Compared with the classical LRU scheme, MT's hit ratios are higher for 8 traces and a bit lower for 5 traces. The highest hit rate improvement of MT occurs at trace *rsrch*, in which MT promotes the hit ratio to 2.6 times of LRU's. Meanwhile, the worst case is a 12.6% reduction for trace *usr*. On average, MT improves the hit ratio by 12% com-

Table 2. The Real-world Traces from Microsoft Research Cambridge Used in our Evaluations

Trace Name	Application Type	Request Count	Read Ratio
usr	User home directories	38,975,431	96.13%
proj	Project directories	40,366,612	5.93%
prn	Print server	17,635,766	19.79%
hm	Hardware monitoring	8,985,487	32.65%
rsrch	Research projects	3,254,278	11.22%
prxy	Firewall/web proxy	22,642,550	3.61%
src	Source control	2,999,229	12.45%
stg	Web staging	6,098,667	31.76%
ts	Terminal server	4,216,457	25.89%
web	Web/SQL server	9,642,398	53.59%
mds	Media server	2,916,662	29.61%
wdev	Test web server	2,654,824	27.3%

pared with LRU. Though the hit rates of MT are only higher than LFU in 4 out of 13 cases, the average hit ratio of MT is only 7.7% lower than LFU. As for ARC and LARC, the average hit ratio of MT is close to them, i.e., 94.9% of ARC's and 94.4% of LARC's. MT exhibits higher hit rates than those of SieveStore in 11 out of 13 cases. The average hit ratio improvement of MT based on SieveStore is 16.1%. When compared with pdLRU, MT has a higher hit rate in all 13 cases and achieves a 30% improvement on average.

Fig. 11 exhibits that the NVM write amounts of MT are always much fewer than those of all the other algorithms except pdLRU for all the traces. Note that for each trace, the NVM write amounts of all the caching algorithms in Fig. 11 have been normalized based on the MT's write amount. Compared with LRU, for example, MT reduces write amounts at most 117.44 times (*rsrch*) and at least 10.24 times (*stg*). The average write amount reduction is 28.76 times. MT reduces the writing pressure of NVM devices by 24.79 times, 25.58 times and 2.643 times, compared with LFU, ARC and LARC, respectively. Compared with SieveStore, the writing reduction of MT reaches 6.75 times on average.

The results plotted in Figs. 10 and 11 confirm that MT is conducive to discovering the long-term hot data, thereby reducing the NVM write amount significantly without unnecessary performance loss.

4.3 Results under different Cache Sizes

The experimental results of the hit rates and the NVM write amounts under different NVM cache sizes driven by the traces *wdev* and *prxy* are given in this part. The hit ratio of MT for the trace *wdev* is higher than that of LRU and is a bit lower for *prxy*. We pick these two different kinds of traces to provide good coverage of different cases.

Fig. 12 and Fig. 13 plot the hit ratios under *wdev* and *prxy* respectively, and the capacity ratios between cache and storage range from 4% to 16%. From both figures, the hit ratios of all the cache algorithms increase along with the ascending cache sizes.

Fig. 12 reveals that the hit rates of MT for trace *wdev* are either the highest or very close to the highest

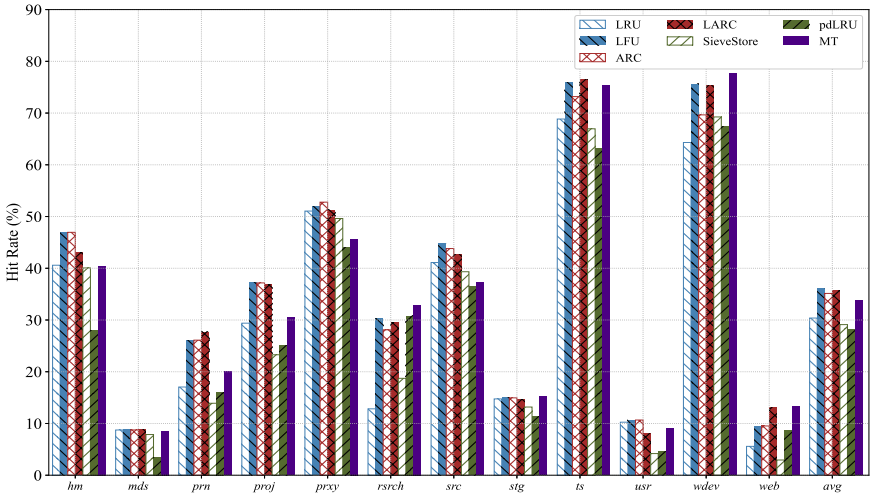

Fig.10. Overall cache hit ratio results.

in all cases. For example, compared with the classical LRU, MT improves the hit ratio by an average of 38.7%. LFU has the best performance except for MT and the average hit ratio of MT increases by 1.74% compared with LFU. For trace *prxy*, the hit rates of MT are slightly lower than the other cache algorithms. Considering the one or two magnitudes lower write amount, such a slight hit rate loss is acceptable for MT.

Fig. 14 and Fig. 15 exhibit the write amounts of the seven caching replacement algorithms. Note that all the results in this figure are normalized based on the write amount of MT when the cache size is 4% of the entire storage.

The results indicate that MT significantly reduces the SSD write loads in all the cases except for pdLRU. For example, when the cache size is 4% of storage capacity, MT reduces the write amounts by a factor of 163.82 compared with LRU, 12.52 compared with LARC and 46.67 compared with SieveStore for *wdev*. Moreover, MT reduces the write amounts by 84.69 compared with LRU, 10.37 compared with LARC and 13.93 compared with SieveStore for *prxy*.

Along with the increase of the cache size, the write amounts of LRU, LFU, ARC and SieveStore keep declining. This trend is attributed by increased hit rates. pdLRU and MT limit the allowed block counts for data updating proportional to the cache size. Therefore, MT's write amount increases as the cache size goes up. In fact, this scheme of MT is more appropriate than the others, because a large cache device can stand more write pressure than a small one.

The case of LARC is more complex. For trace *wdev*, the write amount keeps declining but for trace *prxy*, the trend is the opposite. This is driven by the size of the filter inside LARC which stays larger and generates more updates for trace *prxy*. When the cache size is 16% of the storage capacity, MT still reduces the write amounts by a factor of 8.1 (*wdev*) and 13.93 (*prxy*) compared with LRU. Compared with LARC which achieves the lowest writing amount apart from MT, the average updating amount of MT reduces 6.18 times for *wdev* and 5.99 times for *prxy*. It means that MT has lengthened the device lifetime or decreased energy consumption by 6 times.

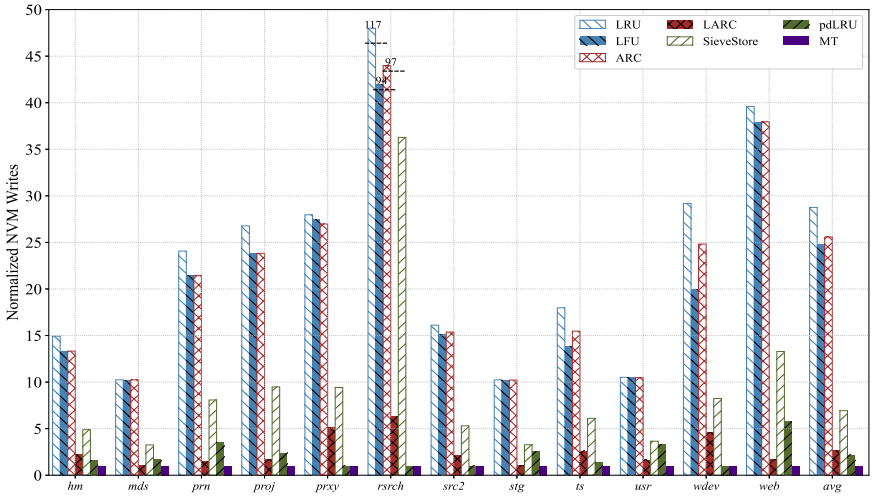

Fig.11. Overall write amount results.

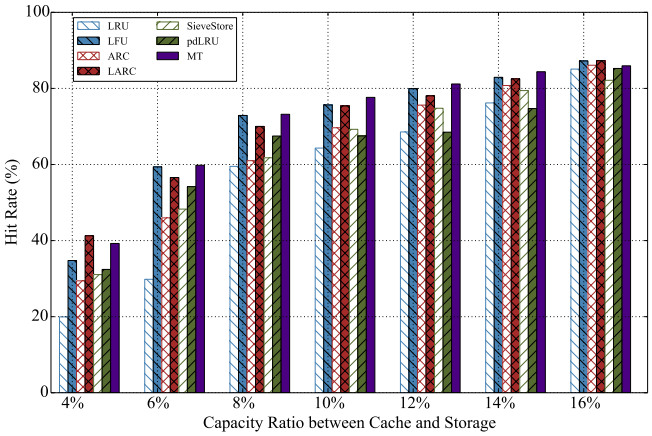

Fig.12. Hit ratios under different cache sizes for *wdev*.

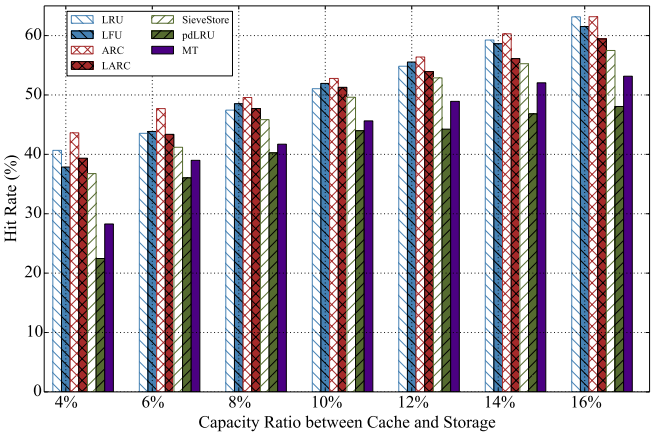

Fig.13. Hit ratios under different cache sizes for *prxy*.

4.4 Quality Analysis of Cached Data

From Subsections 4.2 and 4.3, we can find that MT achieves similar or even higher hit rates with much fewer data updates in many cases. The reason lies in that MT promotes the quality of selected cache blocks by utilizing MT prediction, and thus we do not need to update the cached contents frequently to ensure a high hit rate.

In this part, we will take a deep quality analysis of cached data based on the *wdev* and *prxy* traces from MSR-Cambridge to explain why MT can achieve such good effects. LRU and SieveStore are chosen to be compared with MT since they are good representatives for traditional caching algorithms and writing-optimized caching algorithms. Specifically, we introduce two different metrics in Subsections 4.4.1 and 4.4.2 to measure the cached data quality respectively.

4.4.1 Global Rank of Cached Data

In this part, we use the global rank of the total access count among all the accessed blocks as the metric of long-term hotness. Obviously, the lower the value of a block's global rank is, the more popular the data block

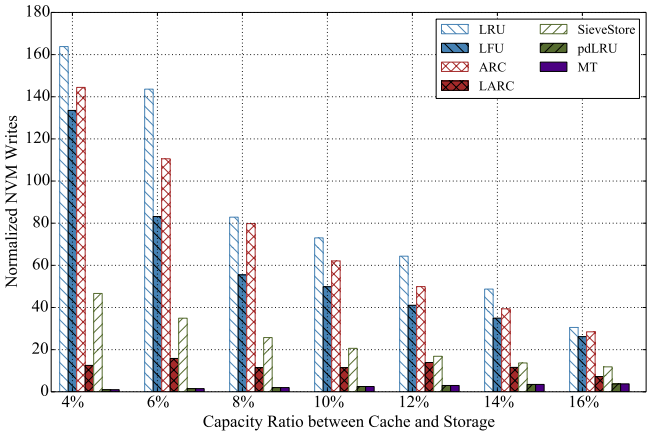

Fig.14. Write amounts under different cache sizes for *wdev*.

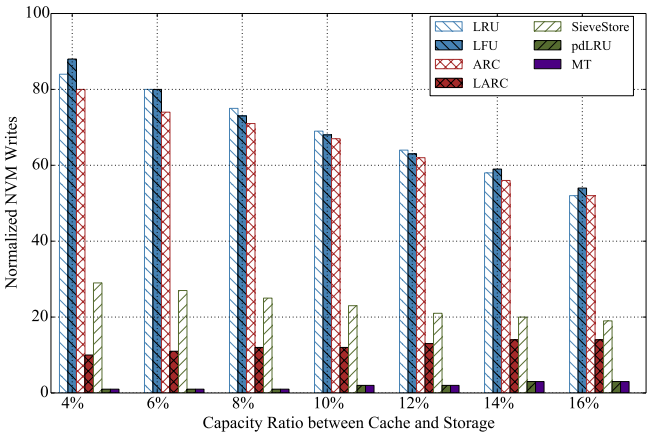

Fig.15. Write amounts under different cache sizes for *prxy*.

is in a long run. For a specified cache algorithm, we can calculate the average global rank of all the ever cached blocks according to this algorithm. A good cache algorithm should lead to a small value of the average global rank, indicating the algorithm is good at identifying long-term hot data for NVM caches.

Based on the traces *wdev* and *prxy*, we first rank all the accessed blocks according to their total access counts. And then we record the global rank value of each cached block that has ever entered the cache, and calculate the average global rank finally. The results of LRU, SieveStore, and MT under the *wdev* and *prxy* traces are shown in Fig. 16. It is obvious that MT achieves the best average global ranks for both traces.

The average global rank of MT is only 18.3% for *prxy*, and those of LRU and SieveStore are 44.4% and 36% respectively. For the *wdev* trace, the value of MT is as small as 6.7%. And the global ranks of LRU and Sievestore are 25.3% and 20.5% respectively. The results indicate that MT can effectively find the long-term hot data, which is a driving force behind MT's high hit rate with its reduced write amount.

4.4.2 *Distribution of Four Quadrants*

In order to maintain a hit rate with fewer NVM writes, we need to achieve two goals, 1) identifying long-term hot data accurately, 2) keeping them for a long time in the NVM-based read cache, but not evicting them too early. Therefore, in this part, we classify all the cached data blocks into four quadrants based on both their popularity (i.e., the total access count) and the count that they have been written into NVM cache repeatedly during the whole trace.

As Fig. 17 shows, the cached blocks located in Q_A have good popularity and they are accessed uniformly without long time absence, and thus they are rarely evicted from the NVM cache. Q_B is for the data that are often evicted from NVM cache because there are often some time periods without any accesses, and then they will be reloaded into the NVM cache due to their high popularity. The other quadrants of Q_C and Q_D contain the unpopular data. In a word, an ideal cache replacement algorithm for NVM-based read cache should lead to a large proportion of Q_A and small proportions of the other three quadrants.

For the *wdev* and *prxy* traces, we perform the above classification, in which the top 20% data blocks are considered as hot and a block is not ideal for cache if the block has been updated in and evicted from cache more than 5 times.

Fig. 18 shows the percentage distributions of the four quadrants for the three cache algorithms under the

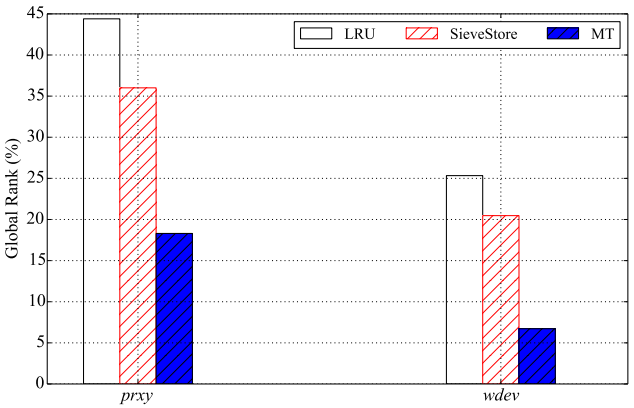

Fig.16. Quality analysis of cached data.

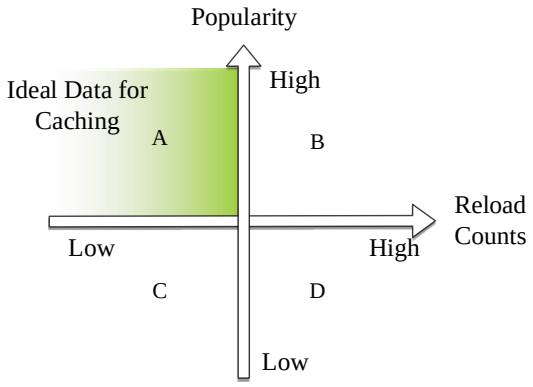

Fig.17. Concept of four quadrants.

two traces.

The Q_A portions of MT are larger for both *prxy* and *wdev* in all cases. Compared with LRU and SieveStore, the Q_A portions of MT reach up to 67.9% (*prxy*) and 87.3% (*wdev*), while LRU only has 0.3% and 1.9%, and SieveStore leads to 17.3% and 17.51%, respectively.

This indicates that when MT limits the data updates for the NVM cache, its decision is wise, i.e., most unpopular data in the long run are kept off from the cache and the best data with both high popularity and stable hotness are selected by MT. What is more, MT protects the hot data well in NVM caches, without loading hot blocks repeatedly. All of these above reasons lead to MT's good effects on both high hit ratios and low data updating amounts.

As Section 1 and Fig. 1 explained, traditional caching algorithms fail to predict the long future of data hotness effectively because they do not utilize enough access information for prediction. The deep quality analysis of cached data verifies the conclusion and the effectiveness of MT in finding long-term hot data.

4.5 I/O Response Time

In order to evaluate the performance of different cache algorithms on practical I/O devices, our simulator supports performing practical I/Os on cache and storage devices, including serving read/write requests and cache replacements. In this part, we will illustrate and analyze the average I/O time of running the whole workloads. We set an 800 GB (gigabyte) PCIe (peripheral component interconnect express) SSD as the read cache and adopt a 2 TB (terabyte) HDD (hard disk drive) as the data storage. The data storage and the cache are associated with a large file on HDD and the other one on SSD, respectively. The requests are sent to devices sequentially and the total time of I/O devices is recorded.

Fig. 19 plots the average I/O response time when running workloads for five times each for the two representative workloads coupled with LRU, SieveStore, and MT, respectively. The average I/O response time of running a whole workload can directly reflect the performance, i.e., a shorter I/O response time indicates a better performance. Among the three algorithms, MT achieves the shortest average I/O response time under both traces, and SieveStore usually leads to the longest I/O time.

Though some data requests need to be loaded from disk several times before it eventually enters cache in MT, it also avoids unnecessary cache misses which significantly reduces the response time.

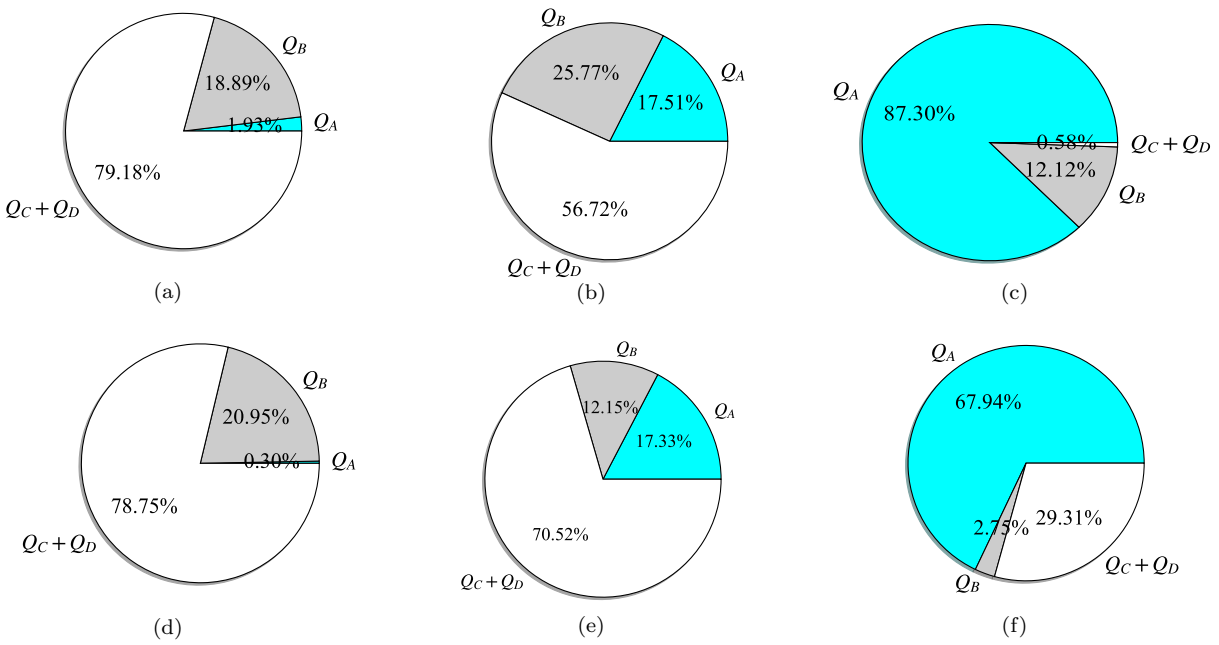

Fig.18. Four-quadrant distribution results under *wdev* and *prxy*. (a)LRU: *wdev*. (b)SieveStore: *wdev*. (c)MT: *wdev*. (d)LRU: *prxy*. (e)SieveStore: *prxy*. (f)MT: *prxy*.

4.6 Impacts of MT Parameters

In this part, we have made a series of experiments based on the traces *wdev* and *prxy* to evaluate the impacts of the key parameters of MT, including the period length and the throttling values of lazy updating.

4.6.1 Period Length

In our implementation of MT, the data updating period is set the same as the macroscopic period for observing data. Therefore when we change the value of the period length, both the data updating frequency and the granularity of recording data access counts are changed at the same time. A too high or too low value of the period length leads to too sparse updating or too few access records to reference, resulting in bad performance. The setting of the period length has a strong influence on the performance of MT.

In this part, we vary the period length from 1,000 to 1,000,000 under *wdev* and *prxy*, and Figs. 20 and 21 reveal the trends of hit ratios and write amounts. Moreover, all the write amounts are normalized by the write amount when the period length is set as 1 000 000. The write amount becomes smaller for larger period length setting because the data updating frequency is lowered. When we increase the period length from 1 000 to 1 000 000 requests, the write amount is slashed by 13.36 times (*wdev*) or 2.31 times (*prxy*).

On the other side, the cache hit ratios usually first increase and then decrease along with larger period length settings. MT makes the judgment on data popularity according to the access counts of the past 10 history periods. When the period length is too small, the lack of enough access records leads to a relatively lower hit rates. When the period length is too large, the data updating frequency is too low to achieve satisfactory performance.

However, the best settings to achieve the highest hit rates for different traces are different. For example, the best period lengths are 100,000 for *wdev* and 10,000 for *prxy*, respectively.

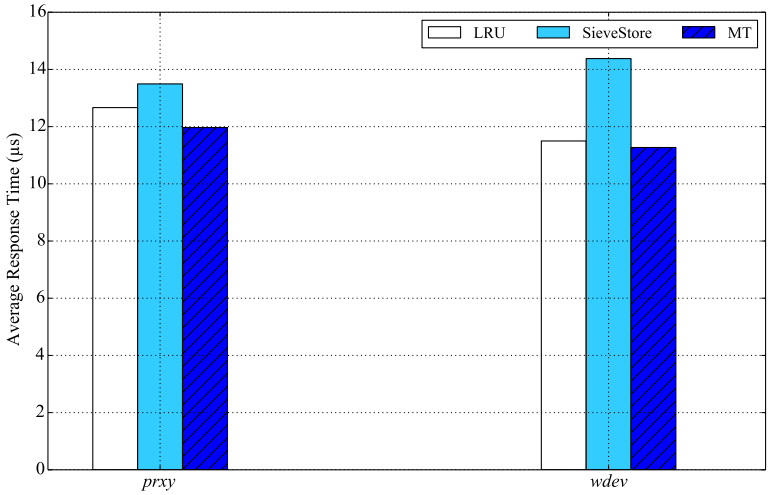

Fig.19. Average I/O response time.

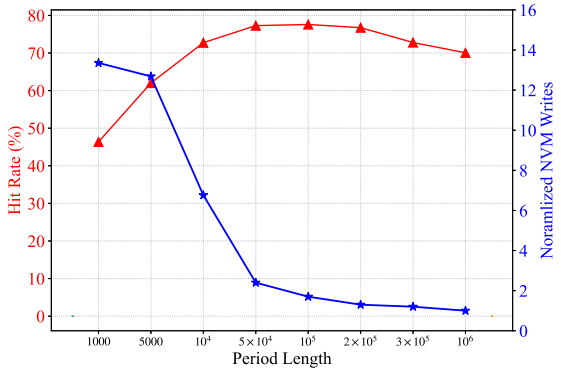

Fig.20. Impacts of period length settings for *wdev*.

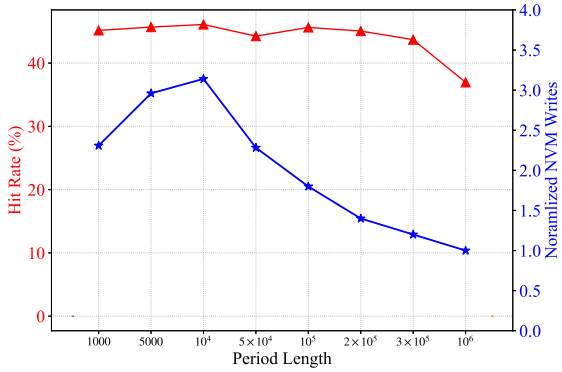

Fig.21. Impacts of period length settings for *prxy*.

4.6.2 *Lazy Updating Throttling*

For the lazy updating throttling (i.e., the maximum updated block count in each round of data updating), a large value leads to more writes, while a small value results in a low cache hit rate due to the lack of new hot data loaded into NVM caches. Figs. 22 and 23 plot the experimental results when the lazy updating throttling value is set in the range of 4% $\sim$ 16% of cache capacity each time. A larger lazy updating throttling value means relaxing the limit on cache replacements, thus increasing both the write amounts and the cache hit rates.

When the lazy updating throttling increases from 4% to 16%, the NVM write amounts increase by 65.6% (*wdev*) and 72.8% (*prxy*) respectively, and the cache hit rate increases only 1.6% for trace *wdev* while the hit rate of 16% is lower than that of 4% for trace *prxy*.

5 Related Work

5.1 Traditional Caching Algorithms.

Traditional caching algorithms are aimed to achieve a high hit rate without considering the data write amount. Except for the classical LRU algorithm, many algorithms have been proposed in the past decades, including FBR (frequency-based replacement)

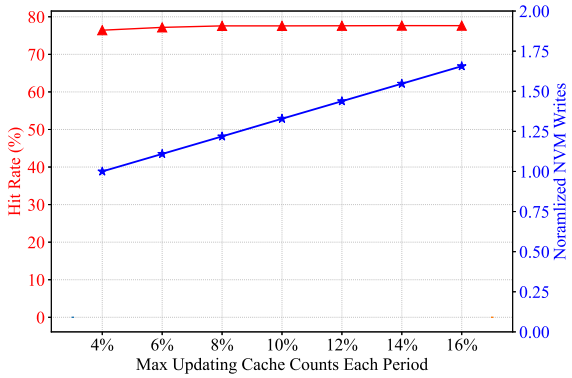

Fig.22. Impacts of throttling for *wdev*.

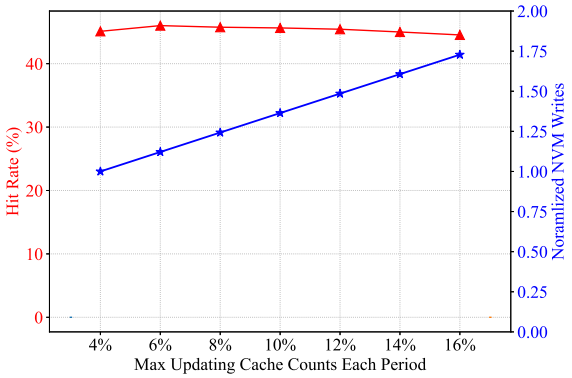

Fig.23. Impacts of throttling for *prxy*.

[16], LRFU (least recently and frequently used) [17], LRU-k [18], MQ (multi-queue) [19], LIRS (low inter-reference recency set) [20], ARC [12], Clock-Pro [21], CAR (clock with adaptive replacement) [22], SxLRU. FBR, LRFU, LRU-k, and MQ are more expensive than LRU. Clock-Pro and CAR extend the CLOCK strategy to trade the cache hit ratio for a low overhead. LIRS considers the access interval to promote the hit rate, while ARC combines the recency and the frequency. SxLRU is a series of caching algorithms including SLRU [23] and many optimized caching algorithms based on SLRU, such as DSB (dueling segmented LRU replacement algorithm with adaptive bypassing) [24] and fixed SLRU (segmented LRU) [25]. The basic idea of SLRU is to divide the cache lines into two lists, the referenced list and the non-referenced list. The non-referenced list is evicted first. Both ARC and SxLRU try to combine the recency and the frequency and ARC has been widely adopted for its performance and adaptability. As a result, ARC was chosen as the representative of traditional caching algorithms in our experiments.

The SSD caches in many industrial products or open-source systems adopt these traditional schemes. For example, Facebook's Flashcache[8] employs FIFO and LRU as its caching schemes, while Linux's dm-cache[9] utilizes MQ [19].

Assuming the disk cache device is DRAM, which has enough write endurance and low write energy consumption, the traditional schemes rarely consider reducing the frequency of updating the cache device and only pursue the short-term hot data for a high hit rate. Thus, this kind of cache algorithms should not be directly used on the emerging NVM devices.

5.2 Caching Schemes with Write Reduction.

In recent years, some studies [26], [27], [28], [29], [30] have focused on applying the SSD-based cache as a popular way of caching data for a high I/O speed. When SSDs are adopted as cache devices, one has to limit the amounts of updated data in SSDs at a low level, because SSDs have very limited total bytes written in their entire life cycle [6]. The aforementioned traditional caching algorithms impose adverse impacts on the lifetime of SSDs as mentioned before. Thus, it becomes indispensable to propose new caching algorithms that extend SSDs' lifetime by limiting the write amount. There have been some cache schemes with write endurance improvement proposed. The existing

[8]Srinivasan M, Saab P. A general purpose, write-back block cache for linux. https://github.com/facebookarchive/flashcache.

[9]Thornber et al. dm-cache. https://en.wikipedia.org/wiki/Dm-cache.

solutions mainly include the following three categories.

Data Fitering Schemes. The first type of schemes adopt a data filtering method to improve the quality of cached data for SSDs and reduce the frequency of updating cached data at the same time.

For example, the Solaris ZFS file system adopts L2ARC to reduce the write amount of SSD-based read caches[10]. L2ARC fills the SSD cache with the contents in the tail of an RAM cache queue. The caching scheme of the RAM queue is ARC [12]. L2ARC selects data cached in SSDs in the same way as that of ARC. In order to reduce the amounts of written data in SSDs, L2ARC periodically fills SSDs (i.e., some hot contents may randomly be ignored) and adopts FIFO (first in first out) as the management scheme for SSDs to reduce the flash write amplification. In fact, L2ARC selects cached data in a sampling manner, thereby leading to unpredictable system performance (i.e., performance may be improved or degraded).

SieveStore [10] and LARC [11] also belong to this category by utilizing some kinds of data filters to select cache targets for SSDs. Some industrial storage products adopt similar caching approaches (e.g., EMC FAST Cache[11], Intel Turbo Memory [31], Netapp's Intelligent Caching[12], and Oracle Exadata's smart flash cache[13]).

However, these algorithms select data based on queueing data by traditional cache schemes which pursue short-term hottest data. The data prediction method is not improved by these schemes to adapt to the long-future prediction requirements of the write-amount-sensitive NVM devices. According to these algorithms, a part of short-term hottest data chosen by traditional prediction methods, instead of long-term hot data are selected to fill NVM caches, leading to unnecessary performance loss.

In fact, our proposed MacroTrend algorithm falls into this category, and thus we compare MacroTrend with the typical SieveStore algorithm in the evaluation part (Section 4). The most significant difference between MacroTrend and other data filtering solutions lies in that MacroTrend makes an effective improvement on predicting long-term hot data while others follow the short-term hot prediction manner of traditional cache algorithms like LRU.

Compared with our previous work [32], this paper provides a thorough analysis of cache algorithms and gives a detailed explanation of why traditional cache algorithms cannot be adapted to NVM devices with write problems. Moreover, a theoretical analysis and a detailed illustration of MacroTrend are given. We also add experiments to analyse the inner mechanism, the I/O response time and the parameter impacts of MacroTrend.

Delayed Eviction Schemes. The second kind of schemes reduce the SSD write amount by extending the staying time of already cached data in SSD-based read cache to an appropriate level to avoid too early eviction of popular data, such as WEC [13] and ETD-Cache (expiration-time driven cache) [33], because a high percentage of cached contents have been found to be evicted too early before generating cache hits. These algorithms can supply protection for the already cached blocks to lower the frequency of cached data updating.

[10]Bitar R. Deploying hybrid storage pools with oracle flash technology and the oracle solaris zfs file system. https://www.oracle.com/technetwork/systems/archive/o11-077-deploying-hsp-487445.pdf.

[11]Emc fast cache: A detailed review. http://www.emc.com/collateral/software/white- papers/h8046-clariion-celerra-unified-fast-cache-wp.pdf.

[12]Woods M. Optimizing storage performance and cost with intelligent caching. http://www.netapp.com/us/media/wp-7107.pdf.

[13]Subramaniam M. Exadata smart flash cache features and the oracle exadata database machine. http://www.oracle.com/technetwork/server- storage/engineered-systems/exadata/exadata-smart-flash- cache-366203.pdf.

However, when the cache replacement happens, they also rely on traditional algorithms like LRU to select data to fill the cache devices. The method proposed in [34] is a special example of delayed eviction schemes. The approach utilizes the out-of-place update mechanism of flash devices. When a page is evicted from the flash cache, the data of the page will still be kept in the device physically for a short time. Therefore, the algorithm allows the user to access the data during this period. The approach is orthogonal to other cache algorithms and can be combined to provide better performance. Moreover, the approach is limited to flash devices.

Container-level Cache Schemes. The third group of cache schemes aims to lengthen the lifetime of SSD caches by reducing the inner write amplification rate of flash chips. When SSDs encounter small random writes, it will cause many additional data movements for garbage collection, i.e., practical write amounts of SSDs are amplified compared with the write operations demanded by the users.

These algorithms manage and replace SSD cached contents in a very large unit (e.g., 128MB), which contains many small cached blocks and is larger than the erase unit size of inner flash chips. Therefore, write amplification is much reduced to lengthen the SSD lifetime and to improve the SSD I/O speed. RIPQ (restricted insertion priority queue) [35], Pannier [36] and SRC (SSD redundant arrays of independent disks as a cache) [37] belong to this kind of container-level cache schemes. These solutions do not have any improvement on data prediction methods, and are only effective for flash-based SSDs but not for other NVM devices. In fact, the idea of container-level cache is orthogonal to our proposed MacroTrend algorithm, i.e., enlarging the cache replacement unit can also be integrated to MacroTrend for further improvement, but it is out of the scope of this paper.

In addition, some other work focuses on reducing the NVM write amount from other aspects. For instance, HEC (high endurance cache) [38] is designed to integrate the cache replacement algorithm with GC (garbage collection) of flash devices. GC needs to copy valid blocks in the victim page, which causes extra write pressure. However, for a flash cache, cold clean data blocks can be directly evicted from cache without extra updates. Based on this observation, the optimized GC mechanism in HEC considers both the number of valid blocks and the hotness of data with the assistance of cache replacement algorithm. However, the algorithm is designed specially for flash devices and needs to change the firmware of flash cache.

It is novel for CacheDedup [39] to integrate the data deduplication techniques into flash cache to reduce the write pressure.

The objective of CloudCache [40] is to improve the performance and lifetime of flash cache in cloud computing by utilizing the reuse working set to predict appropriate cache sizes for each VM (virtual machine). And SHARDS (spatially hashed approximate reuse distance sampling) [41] can reduce the time and space overhead of MRC (miss ratio curve). MRC is an important measurement of the temporal locality of workloads for estimating an appropriate cache size and is constructed by sampling.

In addition, CFLRU (clean first LRU) [42] is a caching algorithm of DRAM on the top of an SSD device, which reduces the writing pressure by evicting the clean pages in DRAM first. The scope of this paper is how to manage the NVM-based read cache, which means that all the mentioned algorithms are operated below DRAM and inside NVM. As a result, CFLRU is orthogonal to MacroTrend and can be utilized as the caching algorithm of memory to reduce the writ-

ing pressure of NVM devices further.

6 Conclusions and Future Work

In this section, we conclude this paper and introduce the practical meaning and future work of our approach.

1) In this paper, we propose a novel MacroTrend prediction framework to keep track of the macroscopic popularity trends of a block through its access count histogram in a history window. Moreover, we develop a new MacroTrend cache replacement algorithm based on the prediction framework to reduce the write amounts of NVM-based read caches significantly with maintaining high hit rates. The experimental results driven by the real-world traces reveal that MacroTrend achieves high hit rates and significantly low writing amounts for NVM-based read caches.

2) MacroTrend has high potencial to be utilized in numberous areas with a huge read I/O requirement, such as machine learning and OLAP (online analytical processing), to boost the system performance and reduce the device replacement cost. Moreover, under the MacroTrend prediction framework, how to pinpoint the long-term hot data accurately is an open problem. In the future, we can incorporate new methods like machine learning to build a more precise relationship between the data access histograms and the long-term data popularity to make further improvement.

References

[1] Leventhal A. Flash storage memory. *Communications of the ACM*, 2008, 51(7): 47-51. DOI: 10.1145/1364782.1364796.

[2] Lee B C, Ipek E, Mutlu O, Burger D. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009, pp.2-13. DOI: 10.1145/1555754.1555758.

[3] Xu C, Niu D, Muralimanohar N, Balasubramonian R, Zhang T, Yu S, Xie Y. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, February 2015, pp.476-488. DOI: 10.1109/HPCA.2015.7056056.

[4] Hosomi M, Yamagishi H, Yamamoto T, Bessho K, Higo Y, Yamane K, ..., Kano H. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. In *IEEE International Electron Devices Meeting, IEDM Technical Digest*, December 2005, pp.459-462. DOI: 10.1109/IEDM.2005.1609379.

[5] Grupp L M, Davis J D, Swanson S. The bleak future of NAND flash memory. In *10th USENIX Conference on File and Storage Technologies*, February 2012, Vol. 7, No.3.2.

[6] Boboila S, Desnoyers P. Write endurance in flash drives: Measurements and analysis. In *8th USENIX Conference on File and Storage Technologies*, February 2010, pp.115-128.

[7] Xia F, Jiang D, Xiong J, Sun N. Hikv: A hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference*, July 2017, pp.349-362.

[8] Jiang L, Zhang Y, Yang J. ER: elastic RESET for low power and long endurance MLC based phase change memory. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, July 2012, pp.39-44. DOI: 10.1145/2333660.2333672.

[9] Hirofuchi T, Takano R. RAMinate: Hypervisor-based virtualization for hybrid main memory systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, October 2016, pp.112-125. DOI: 10.1145/2987550.2987570.

[10] Pritchett T, Thottethodi M. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, June 2010, pp.163-174. DOI: 10.1145/1815961.1815982.

[11] Huang S, Wei Q, Feng D, Chen J, Chen C. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage*, 2016, 12(2): 1-24. DOI: 10.1145/2737832.

[12] Megiddo N, Modha D S. ARC: A self-tuning, low overhead replacement cache. In *2rd USENIX Conference on File and Storage Technologies*, March 2003, pp.115-130.

[13] Chai Y, Du Z, Qin X, Bader D A. WEC: Improving durability of SSD cache drives by caching write-efficient data. *IEEE Transactions on Computers*, 2015, 64(11): 3304-3316. DOI: 10.1109/TC.2015.2401029.

[14] Wang L, Zhan J, Luo C, Zhu Y, Yang Q, He Y, ..., Qiu B. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, February 2014, pp.488-499. DOI: 10.1109/HPCA.2014.6835958.

[15] Narayanan D, Donnelly A, Rowstron A. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage*, 2008, 4(3): 1-23. DOI: 10.1145/1416944.1416949.

[16] Robinson J T, Devarakonda M V. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, April 1990, pp.134-142. DOI: 10.1145/98457.98523.

[17] Lee D, Choi J, Kim J H, Noh S H, Min S L, Cho Y, Kim, C S. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1999, pp.134-143. DOI: 10.1145/301453.301487.

[18] O'neil E J, O'neil P E, Weikum G. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 1993, 22(2): 297-306. DOI: 10.1145/170036.170081.

[19] Zhou Y, Chen Z, Li K. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 2004, 15(6): 505-519. DOI: 10.1109/TPDS.2004.13.

[20] Jiang S, Zhang X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 2002, 30(1): 31-42. DOI: 10.1145/511399.511340.

[21] Jiang S, Chen F, Zhang X. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *USENIX Annual Technical Conference*, April 2005, pp.323-336.

[22] Bansal S, Modha D S. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies*, March 2004, pp.187-200.

[23] Karedla R, Love J S, Wherry B G. Caching strategies to improve disk system performance. *Computer*, 1994, 27(3): 38-46. DOI: 10.1109/2.268884.

[24] Gao H, Wilkerson C. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *Proc. of the 1st JILP Workshop on Computer Architecture Competitions*, June 2010.

[25] Morales K, Lee B K. Fixed segmented LRU cache replacement scheme with selective caching. In *2012 IEEE 31st International Performance Computing and Communications Conference*, December 2012, pp.199-200. DOI: 10.1109/PCCC.2012.6407712.

[26] Saxena M, Swift M M, Zhang Y. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, April 2012, pp.267-280. DOI: 10.1145/2168836.2168863.

[27] Kgil T, Roberts D, Mudge T. Improving NAND flash based disk caches. In *International Symposium on Computer Architecture*, June 2008, pp.327-338. DOI: 10.1109/ISCA.2008.32.

[28] Yang Q, Ren J. I-CASH: Intelligently coupled array of SSD and HDD. In *IEEE 17th International Symposium on High Performance Computer Architecture*, February 2011, pp.278-289. DOI: 10.1109/HPCA.2011.5749736.

[29] Ren J, Yang Q. A new buffer cache design exploiting both temporal and content localities. In *IEEE 30th International Conference on Distributed Computing Systems*, June 2010, pp.273-282. DOI: 10.1109/ICDCS.2010.26.

[30] Zhang Y, Soundararajan G, Storer M W, Bairavasundaram L N, Subbiah S, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Warming up storage-level caches with bonfire. In *the 11th USENIX Conference on File and Storage Technologies*, February 2013, pp.59-72.

[31] Matthews J, Trika S, Hensgen D, Coulson R, Grimsrud K. Intel turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *ACM Transactions on Storage*, 2008, 4(2): 1-24. DOI: 10.1145/1367829.1367830.

[32] Bao N, Chai Y, Qin X. A write-efficient cache algorithm based on macroscopic trend for NVM-based read cache. In *2019 Design, Automation and Test in Europe Conference and Exhibition*, March 2019, pp.1245-1248. DOI: 10.23919/DATE.2019.8715276.

[33] Dai N, Chai Y, Liang Y, Wang C. ETD-Cache: An expiration-time driven cache scheme to make SSD-based read cache endurable and cost-efficient. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, May 2015, No.26. DOI: 10.1145/2742854.2742881.

[34] Xia Q, Xiao W. High-performance and endurable cache management for flash-based read caching. *IEEE Transactions on Parallel and Distributed Systems*, 2016, 27(12): 3518-3531. DOI: 10.1109/TPDS.2016.2537822.

[35] Tang L, Huang Q, Lloyd W, Kumar S, Li K. RIPQ: Advanced photo caching on flash for Facebook. In *the 13th USENIX Conference on File and Storage Technologies*, February 2015, pp.373-386.

[36] Li C, Shilane P, Douglis F, Wallace G. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference*, November 2015, pp.50-62. DOI: 10.1145/2814576.2814734.

[37] Oh Y, Lee E, Hyun C, Choi J, Lee D, Noh S H. Enabling cost-effective flash based caching with an array of commodity SSDs. In *Proceedings of the 16th Annual Middleware Conference*, November 2015, pp.63-74. DOI: 10.1145/2814576.2814814.

[38] Yang J, Plasson N, Gillis G, Talagala N, Sundararaman S, Wood R. HEC: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, June 2013, No.10. DOI: 10.1145/2485732.2485743.

[39] Li W, Jean-Baptise G, Riveros J, Narasimhan G, Zhang T, Zhao M. CacheDedup: In-line deduplication for flash caching. In *the 14th USENIX Conference on File and Storage Technologies*, February 2016, pp.301-314.

[40] Arteaga D, Cabrera J, Xu J, Sundararaman S, Zhao M. CloudCache: On-demand flash cache management for cloud computing. In *the 14th USENIX Conference on File and Storage Technologies*, February 2016, pp.355-369.

[41] Waldspurger C A, Park N, Garthwaite A T, Ahmad I. Efficient MRC construction with SHARDS. In *the 13th USENIX Conference on File and Storage Technologies*, February 2015, pp.95-110.

[42] Park S Y, Jung D,Kang J U, Kim J S, Lee J. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, October 2006, pp.234-241. DOI: 10.1145/1176760.1176789.

Ning Bao is a Ph.D. candidate in the Department of Computer Science at Renmin University of China, Beijing. She received the B.S. degree in Computer Science from Renmin University of China, Beijing, in 2015 and the M.S. degree in Computer Science from National University of Singapore in 2018. Her research interests include emerging storage techniques, cloud computing, and caching systems.

Yun-Peng Chai received his B.E. and Ph.D. degrees in computer science and technology from Tsinghua University, Beijing, in 2004 and 2009, respectively. He is currently an associate professor at School of Information, Renmin University of China, Beijing. His research interests include emerging storage techniques, cloud computing, and distributed systems. He is a member of CCF.

Xiao Qin received his B.S. and M.S. degrees in computer science from the Huazhong University of Science and Technology, Wuhan, in 1996 and 1999, respectively, and his Ph.D. degree in computer science from the University of Nebraska, Lincoln, in 2004. He is currently a professor of computer science at Auburn University, Auburn. His research interests include parallel and distributed systems, real-time computing, storage systems, fault tolerance, and performance evaluation. He received an NSF (National Science Foundation of USA) CAREER Award in 2009. He is a senior member of IEEE and the IEEE Computer Society.

Chuan-Wen Wang received her B.S. and M.S. degrees in computer science from Renmin University of China, Beijing, in 2017 and 2020, respectively. She is going to pursue Ph.D. study at the Chinese University of Hong Kong. Her research interests include distributed systems, storage systems and cloud resource management.

Reply Letter for Reviewer 1

Comments to the Author:
The revised manuscript addressed all my previous comments well and there is no further comments/suggestions. I suggest accepting the paper for publication.

Thank you very much the valuable comments which enhanced our manuscript significantly.

Reply Letter for Reviewer 2

Comments to the Author:
This paper introduced an NVM-based read cache algorithm with a close hit ratio and higher write efficiency compared with traditional cache algorithms (LRU, LFU, ARC, etc.). The authors observed that previous cache algorithms either lack proper abstraction in prediction or frequently update data, resulting in a large number of updates in a read cache. To address this problem, this paper uses a hash table to collect access count histograms in some past periods for each block. The lazy update strategy is used to periodically update the cache according to the access count histograms of each block. Extensive evaluations show that the proposed cache algorithm can reduce the number of NVM writes by 28.76 times on average.

The evaluation is comprehensive. The quality analysis of cache data well explains the performance efficiency.

The main overhead of this algorithm comes from the update operation. However, these operations are processed in backgrounds, which is not clear.

CQ and NQ are both temporary data structures. Does this mean that the system needs to temporarily allocate memory for every update operation? If it is, would this seriously affect the performance of the algorithm? If not, a general introduction becomes much better to show how this algorithm manages those temporary data structures.

We greatly appreciate this reminder. The memory management of CQ and NQ for the update operations has been elaborated in Section 3.5.3 of the revised version of our manuscript.

3.5.3 Space Overhead

…

CQ stores the candidate blocks which are a partition of the history table. And NQ stores the metadata of cached blocks. As a result, the space overhead

of NQ and CQ is limited and MacroTrend keeps the memory for CQ and NQ to avoid the frequent memory allocation operations.

…

The number of blocks in CQ is the update throttling and the size of NQ is the cache capacity. As a result, the space consumption of both the two data structures is stable. Therefore, MacroTrend keeps the allocated memory of CQ and NQ in the long run to avoid frequent memory allocation and release operations, which may introduce additional overheads. If the size of the two queues changes (e.g., the updating throttling is modified by the user), MacroTrend will adjust the allocated memory space accordingly.

Reply Letter for Editor,

非常感谢您细致全面的建议，对我们提升论文的质量和可读性有很大的帮助。

我们已经按照所有的标注、论文模板和自检表对论文格式和写作进行了全面的检查和修改。所有的图片和图片标题的格式、参考文献的格式和论文中的主谓语单复数等格式都严格按照要求进行了修改。论文的Introduction和Conclusion部分也按照要求进行了调整和修改，Introduction中的部分内容被调整到了第二章。此外，我们之前在DATE19发表的论文也加入到了参考文献，并且在第五章相关工作中和本论文进行了对比。