

A Hint Frequency Based Approach to Enhancing the I/O Performance of Multilevel Cache Storage Systems

Xiao-Dong Meng, *Member, CCF*, Chen-Tao Wu*, *Member, CCF, IEEE*, Min-Yi Guo, *Senior Member, IEEE*, Jie Li, *Senior Member, ACM, IEEE*, Xiao-Yao Liang, Bin Yao, and Long Zheng

Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

E-mail: mengxiaodong1985@sjtu.edu.cn; {wuct, guo-my, lijie, liang-xy, yaobin, lzheng}@cs.sjtu.edu.cn

Received April 5, 2016; revised December 29, 2016.

Abstract With the enormous and increasing user demand, I/O performance is one of the primary considerations to build a data center. Several new technologies in data centers, such as tiered storage, prompt the widespread usage of multilevel cache techniques. In these storage systems, the upper level storage typically serves as a cache for the lower level, which forms a distributed multilevel cache system. However, although many excellent multilevel cache algorithms have been proposed to improve the I/O performance, they still have potential to be enhanced by investigating the history information of hints. To address this challenge, in this paper, we propose a novel hint frequency based approach (HFA), to improve the overall multilevel cache performance of storage systems. The main idea of HFA is using hint frequencies (the total number of demotions/promotions by employing demote/promote hints) to efficiently explore the valuable history information of data blocks among multiple levels. HFA can be applied with several popular multilevel cache algorithms, such as Demote, Promote and Hint-K. Simulation results show that, compared with original multilevel cache algorithms such as Demote, Promote and Hint-K, HFA can improve the I/O performance by up to 20% under different I/O workloads.

Keywords storage system, multilevel cache, hint, I/O performance

1 Introduction

In big data era, I/O performance is still the bottleneck of data processing in many fields^[1–10]. In large data centers, heterogeneous storage devices cooperate together to accelerate the I/O processing. Typically, the storage devices in the upper level serve as caches for the lower level, which forms a distributed multilevel cache system. In recent years, multilevel caches have received more attention due to the following reasons.

- *High I/O Performance.* By aggregating heterogeneous storage devices together, multilevel caches

achieve higher I/O performance compared with using single level caches separately^[11–13].

- *Low Monetary Cost.* Typically, multilevel cache policies can provide global management on all cache levels without any manual operations^[14]. It can sharply decrease the monetary cost.

- *High Flexibility.* Multilevel caches are widely used in many scenarios, such as in traditional client-server model^①^[15–16], networked storage^②^[12–13] and hybrid storage^③^[17].

In the past two decades, many classic multilevel cache solutions were proposed to improve the I/O per-

Regular Paper

This work was supported by the National Basic Research 973 Program of China under Grant No. 2015CB352403, the National High Technology Research and Development 863 Program of China under Grant No. 2015AA015302, the National Natural Science Foundation of China under Grant Nos. 61332001, 61303012, 61572323 and 61628208, the Scientific Research Foundation for the Returned Overseas Chinese Scholars, and the CCF-Tencent Open Fund.

*Corresponding Author

① The clients provide the first-level caches, and the servers supply the second-level caches. In some scenarios, the application servers provide the first-level caches, and the storage servers supply the second-level caches.

② Metadata servers and data storage servers offer the upper level and the lower level caches, respectively.

③ Fast and slow storage devices (e.g., DRAM and flash) serve as the upper level and the lower level caches, respectively.

©2017 Springer Science + Business Media, LLC & Science Press, China

formance of storage systems. One of the most effective approaches is making the cache exclusively among different hierarchies, where hints are well utilized to identify hot data blocks. These hints can provide a global view of a storage system, which can be optimized via general/specific replacement strategies.

According to the roles in different scenarios, hints can be divided into three categories, demote hints, promote hints and application hints. A demote hint^[12] is a flag to mark a demotion operation, which occurs when a data block is evicted from an upper level to a lower level. Similarly, a promote hint^[13] is used to identify a promotion, which is an opposite operation of demotion. Except for demote and promote hints, an application hint^[18-19] is computed by well-defined formula(s) and is evaluated by the access patterns of I/O workloads. A hybrid hint is a flag containing information of a data block from multiple aspects such as demotion, promotion or application access patterns. In many previous literatures^[20-22], hybrid hints are demonstrated to be an efficient way to enhance the performance significantly.

However, existing multilevel cache algorithms still have potential to be improved. Most multilevel cache algorithms^[12-13,19] use hints to store the concentrated information of data blocks. The demote and promote hints based approaches only record the latest hint operations, thereby valuable history hint information is ignored. And the application hints are usually dependent on a specified application, which is not general to all scenarios. Some algorithms^[22-23] keep the latest multiple steps hint information, but they are insufficient to describe the status of data blocks. The detailed illustration is given in Subsection 2.3.

To address the above problems, in this paper, we propose a novel hint frequency based approach, HFA, which is an efficient multilevel cache scheme to enhance the I/O performance. The main idea of HFA is using rich history hint information (e.g., hint frequencies) to efficiently identify hot data blocks.

Our contributions include the following aspects.

- We propose a novel hint frequency based approach, HFA, using history hint information of data blocks, which efficiently enhances the I/O performance of storage systems.
- HFA can cooperate with several famous multilevel algorithms, such as Demote, Promote and Hint-K.
- We implement HFA by combining with Demote, Promote and Hint-K algorithms. Simulation results

demonstrate that HFA can enhance the cache performance under various I/O workloads.

The rest of this paper continues as follows. Section 2 briefly overviews related work and our motivation. In Section 3, we illustrate the design, model and replacement policies of HFA. Section 4 illustrates how HFA collaborates with other popular algorithms. In Section 5, we analyze the simulation results by using various multilevel cache approaches. Finally Section 6 concludes the paper.

2 Related Work and Our Motivation

Many classic cache schemes have been proposed to improve the I/O performance over the past several decades, which can be classified into two main categories: single level caches and multilevel caches. In this section, we briefly introduce the state-of-the-art algorithms and our motivation of this work.

2.1 Single Level Cache Algorithms

LRU^[24] is widely used in buffer cache management. Since the 1990s, many LRU variants aim to improve the performance of single level caches. Due to page limit, here we only list some famous LRU-based algorithms: FBR^[25], 2Q^[26], LRU-K^[27-28], UBM^[29], LRFU^[30], LIRS^[31], ARC^[32], CAR^[33], SPCC^[34], SARC^[35], AMP^[36], DULO^[37], CLOCK-Pro^[38], WOW^[39], RACE^[40], STOW^[41].

2.2 Multilevel Cache Algorithms

Quite a few multilevel cache algorithms emerge to improve the aggregate I/O performance of distributed systems as summarized in Table 1. As introduced in Section 1, hints are widely used for cooperative cache for heterogeneous storage devices. Based on the usage of hints, we divide these cache algorithms into the following five classifications: multilevel algorithms without hints, those with demote hints, those with promote hints, those with application hints, and those with hybrid hints.

Without Hints. MQ^[15] concentrates three properties for a good second-level buffer cache: minimal lifetime, frequency based priority, and temporal frequency to efficiently manage the second-level buffer cache.

With Demote Hints. The demote algorithm^[12] first uses demote hints to describe evicted data information from the upper cache level, which makes caches exclusive. Through demote hints, the dynamic behavior of

evicted caching data is well captured, which increases the cache hit rate and decreases the average latency in different applications. From then on, demote hints have been extensively used in multilevel cache solutions. X-RAY^[42] is another sample which uses demote hints in disk arrays. It gives the array cache the data information on the content through file-node operations and writes log updates. Other cache policies combine with demote hints and the analysis of access patterns to enhance the cache performance, such as EV^[43], GL-MQ^[16], uCache^[44].

Table 1. Different Policies in Cache Algorithm Design

Cache Algorithm	Cache Level	WH	DH	PH	AH	HH
MQ	Last level	✓	×	×	×	×
DEMOTTE	Last level	×	✓	×	×	×
EV	Last level	×	✓	×	×	×
GL-MQ	Last level	×	✓	×	×	×
ULC	All levels	×	✓	×	✓	✓
X-RAY	Last level	×	✓	×	×	×
uCache	All levels	×	✓	×	×	×
TQ	Last level	×	×	×	✓	×
Karma	All levels	×	✓	✓	×	✓
PROMOTE	All levels	×	×	✓	×	×
MC ²	All levels	×	✓	✓	✓	✓
CLIC	All levels	×	×	×	✓	×
C-Hint	All levels	×	×	×	✓	×
Hint-K	All levels	×	✓	✓	×	✓
UARC	All levels	✓	×	×	×	×
HFA	All levels	×	✓	✓	×	✓

Note: WH: without hints. DH: using demote hint. PH: using promote hints. AH: using application hints. HH: using hybrid hints. All levels: at least perform well in a two-level cache hierarchy.

With Promote Hints. The promote^[13,45] algorithms are proposed when hot data needs to be promoted to the higher cache level. Different from demote hints, promotion may cross layers and is decided by more factors (like cache size and hint frequency). Promote policy achieves additional better performance when combined with the ARC algorithm^[32].

With Application Hints. TQ^[18] is proposed to improve the efficiency of the second-level cache by using write hints. According to the access patterns in real database applications, three write hints (SYNCH, REPLACE and RECOV) are used. These hints provide strong indications to show the access patterns of current state and near future, which can manage the second cache level effectively. CLIC^[19] uses dynamic

application hints in various database applications to dramatically describe the hotness of data blocks, and has achieved good results. C-Hint^[46] is a client-assisted cache management approach. It delivers sustainable high cache hit rate with limited cache capacity by analysing access history reports from clients.

With Hybrid Hints. Hybrid hints are appeared in ULC^[47] algorithm. By combining with demote hints and application controlled hints (RETRIEVE), ULC effectively exploits hierarchical locality in multilevel cache. Karma^[20] uses demote hints for exclusivity, and adds two additional operations (READ and READ-SAVE), which can be considered as promotions. Based on the Karma algorithm, MC²^[21] presents a solution for multiple clients with approximate application hints. A state-of-the-art implementation combining Karma and MC² is presented in [48]. Hint-K^[22-23] uses the k -step hint history information of both demote and promote operations, which can efficiently describe the activeness of data blocks. In addition, with the development of tiered storage^[49], unified single level cache algorithms (e.g., UARC^[17]) are another way to provide a comprehensive solution for hybrid memory systems.

2.3 Our Motivation

Nowadays most commercial storage servers use a large storage cache to speed up I/O processing, where hint-based cache algorithms are primary solutions to smooth the gap among heterogenous storage devices^[50-51]. Typically, a general purpose of these solutions is investigating history hint information to efficiently identify hot data blocks. They ignore some important history hint information, such as hint frequency and temporal locality, which motivates us to propose a new solution to enhance the I/O performance of storage systems.

We first study the effectiveness of hint frequency. Hint frequency is the total number of demotions/promotions (by employing demote/promote hints) in a period. The plots in Fig.1(a) show the number of block read hits of 200 pairs of sample blocks from nine traces. Each trace is simulated on a three-level cache with the cache space ratio of 1:2:4, and each kind of bar presents a hint frequency of three levels. The access frequencies of blocks with one demotion/promotion are normalized to 100%. We can see from Fig.1(a) that the blocks with higher hint frequency tend to be visited more frequently for all traces. It dedicates that the higher hint frequency has more important impacts on the access frequencies of data blocks.

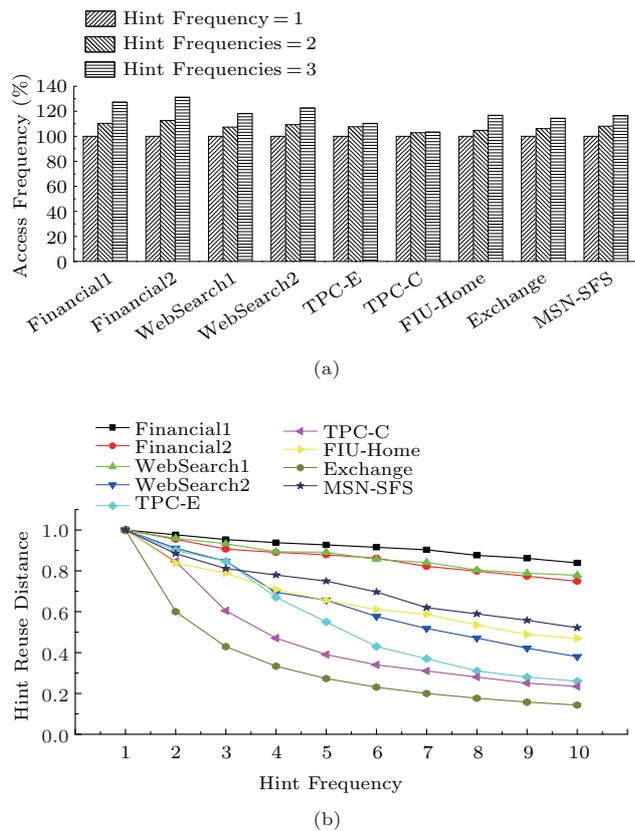


Fig.1. Relationship among hint frequency, access frequency, and hint reuse distance. Nine traces are tested on a three-level cache. The cache space ratio of each level is 1 : 2 : 4. The demote and promote algorithms^[12-13] are used in each cache level for comparison. (a) Access frequency of different traces. (b) Hint reuse distance.

The temporal locality of buffer cache accesses describes the characteristics of traces, which is then used to design replace algorithms to manage buffer caches^[52]. Similarly, we need to analyze the access patterns of demotions/promotions, and design a replacement algorithm based on those patterns. We use hint reuse distance to observe the temporal locality of the traces. Hint reuse distance is the number of distinct accesses between a demote operation and a read miss (or a promote) operation to the same block. For example, in a cache level L_i , if five distinct blocks are visited during the period between a demotion and later a promotion of a block x , then the hint reuse distance of block x is 5.

Fig.1(b) compares the temporal locality of nine traces on a cache that uses the same configuration with that in Fig.1(a). It shows the average hint reuse distance for the blocks grouped by the hint frequency from 1 to 10. The highest value of hint reuse distance for each trace is normalized to 100%. Obviously, for all nine traces, the blocks with higher hint frequency usually

have a smaller hint reuse distance, despite that traces vary in terms of hint frequency sensitivity.

These two observations drive us to develop a practical approach to freeze or promote the blocks with higher hint frequencies in a certain cache level in order to improve the overall cache hit ratio.

3 Hint Frequency Based Approach (HFA)

In this section, we present the design and replacement policies of our hint frequency based approach, HFA. To facilitate the discussion, we summarize the symbols used in this paper in Table 2.

Table 2. Notations

Parameter	Description
n	Number of cache levels
x	Random block in a multilevel cache
L_i ($1 \leq i \leq n$)	The i -th cache level (the first level is L_1 and the last level is L_n)
D_i/P_i	Demotion from L_{i-1} to L_i /promotion from L_{i+1} to L_i
$D_i(x)/P_i(x)$	Number of demotions/promotions of block x in L_i
$hits(x)$	Number of cache hits on block x in a cache level
T	Time slot
t_1	Aggregate duration between demotions and promotions of a block
t_2	Duration that a block resides in a cache level
k	Steps to record blocks' demotion and promotion history
δ	Hotness degradation value
$F_i(x)$	Hint frequencies of block x in L_i
F_{\min}	Threshold hint frequency which determines a queue for a block insertion
$H_i(x)$	Hotness value of block x in L_i
S	Size of cache space exchanged between the two queues
N_C/N_H	Number of cold/hot blocks in Q_1/Q_2
$t_C(x)/t_H(x)$	Resident duration of cold/hot blocks in Q_1/Q_2

3.1 Overview

Our design is to improve I/O performance by investigating some potential history hint information of multilevel cache systems, such as hint frequencies. To achieve this goal, we propose a novel hint frequency based approach (HFA). As shown in Fig.2, the cache model of HFA consists of n cache levels (L_1, L_2, \dots, L_n), excluding the first client level and the last storage de-

vice level. In this model, D_i delegates a demotion from L_{i-1} to L_i , and P_i stands for a promotion from L_{i+1} to L_i . $D_i(x)$ and $P_i(x)$ are used to represent the corresponding demotes/promotes for a data block x . In this model, hot blocks are encouraged to be promoted to the upper level, and cold blocks are evicted via demote operations.

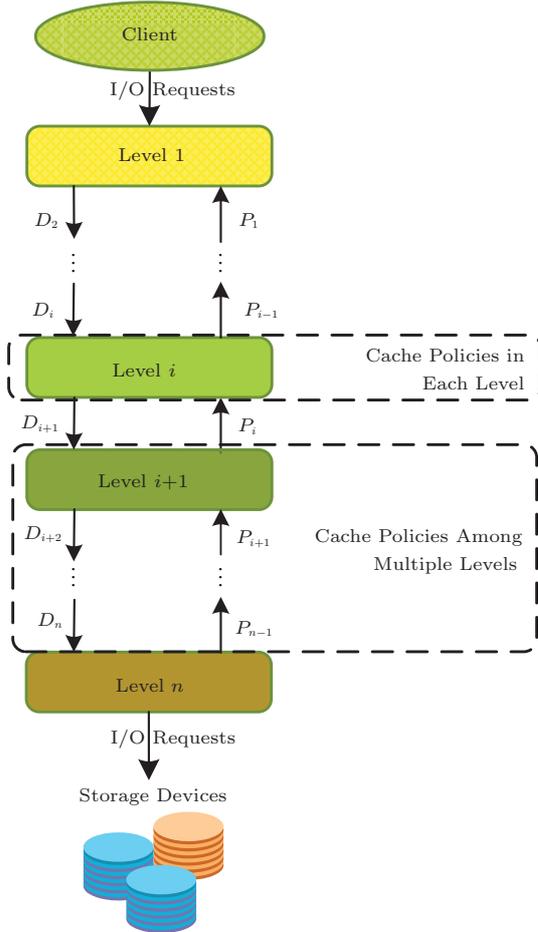


Fig.2. Multilevel cache model of HFA. It consists of n levels, with one cache in each level. Demotions (D_i , $2 \leq i \leq n+1$) and promotions (P_i , $1 \leq i \leq n$) are communicated among different cache levels. D_1 , P_0 , D_{n+1} and P_n are invalid in this model.

3.2 Cache Policies in Each Level

HFA is a comprehensive multilevel cache solution, which can combine with demote/promote hints. It utilizes the hint frequencies to identify hot data blocks. HFA has two main components.

- *Cache Policies in Each Level.* Different from typical multilevel cache algorithms, HFA uses two queues in each level. One queue is used to store data blocks with high hint frequencies sorted by hotness, and the other queue contains data blocks with low hint frequencies.

- *Demotion/Promotion Policies Among Multiple Levels.* In HFA, the demotion/promotion policies add hint frequencies as an important aspect to decide whether a data block is demoted/promoted.

To design an effective cache policy in each level, first we need to efficiently identify hot data blocks. Second, a proper replacement policy should be provided to process the I/O requests.

3.2.1 Identification of Hot Blocks

In HFA, the hotness of a random block x in cache level i is decided by both hint frequency and hit ratio, as calculated by:

$$H_i(x) = \frac{F_i(x)}{t_1 + t_2} + \frac{hits(x)}{t_2}. \quad (1)$$

In (1), t_1 and t_2 present the periods before and after block x is loaded in cache i , respectively. $F_i(x) = D_i(x) + P_i(x)$, is the number of demotions and promotions of block x . And $hits(x)$ is the number of cache hits on block x . Thus, we say a block is hot when it is moved between cache levels and(or) accessed frequently.

As LRU-K^[27] and Hint-K^[22] use k steps to record history information, HFA takes a long time interval ($t_1 + t_2 = kT$, k is an integer and T is a pre-defined time slot) to accurately evaluate the hint frequency value of data blocks. We set a hotness degradation value δ , and aggregate the hint frequency of block x as:

$$F_i(x) = \sum_{j=0}^k \delta^{j-1} F_i^{j-1}(x). \quad (2)$$

Consequently, (1) changes to

$$H_i(x) = \frac{\sum_{j=0}^k \delta^{j-1} F_i^{j-1}(x)}{kT} + \frac{hits(x)}{t_2}. \quad (3)$$

3.2.2 Replacement Policy in Dual Queues

Hotness contains history information, but it is not efficient if we manage caches by simply sorting and replacing blocks by the hotness value. Considering observation 1 and observation 2, a new allocated block has the lowest priority, compared with the migrated and hit blocks. The new allocated blocks are replaced rather than the others when a queue is full. Consequently, in level i , the hotness value of a new allocated block x is set to zero ($H_i(x) = 0$). Block x has a low hotness in the cache queue and is going to be demoted to the lower level repeatedly, regardless of potential intensive accesses on x in the near future. x becomes hot only after it travels inter-cache levels until it is set to be a high hint frequency value. To utilize the property of

hint frequency as we showed in the motivation to improve the cache performance, it calls a sophisticated design of the replacement policy.

As shown in Fig.3, there are two queues in HFA in each level. One is a queue (Q_1), which stores the data blocks with a high hotness value and sorts blocks by the hotness values. According to the observation from Fig.1(b), the blocks with a higher hint frequency value tend to be reused in a near future. Therefore, we set the other queue (Q_2), which saves the blocks with a low hotness value, but sorts the blocks by their hint frequencies. Q_1 has a higher priority than Q_2 , that is, the evicted block from Q_1 will be inserted to Q_2 , instead of erasing or demoting it to a lower level. As a result, blocks in Q_1 can resist in the cache longer.

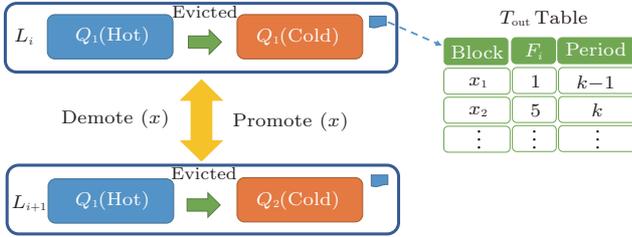


Fig.3. HFA dual-queue structure.

Meanwhile, there are two types of blocks in Q_2 which can be promoted to Q_1 . One is the blocks which are frequently visited. The other is the blocks whose hint frequency is higher than a threshold, because the increment on the value of hint frequency needs the movements of the block between cache levels. But the promoting process from Q_2 to Q_1 introduces the latency to the blocks with potential intensive accesses in the near future. To reduce the latency, the blocks in Q_2 will also be promoted to Q_1 after a hit.

The cache management of Q_2 can apply any proposed multilevel cache policy for single queue, such as Demote^[12], Promote^[13] and Hint-K^[22]. Therefore, the management policy of Q_2 is not mentioned in the following paragraphs.

According to (2), hint frequencies are dynamic. To sort blocks by hint frequencies, in each level, HFA maintains a T_{out} table to record the hint frequency of the recent demoted/promoted blocks from a level (only cross cache level operations are considered; data blocks moving between Q_1 and Q_2 in a same cache level are not recorded in T_{out}). The detail usage of T_{out} is introduced in Subsection 3.2.3. The size of table T_{out} is fixed, which

is set to 0.1% of the corresponding cache size in each level by default.

The table T_{out} organizes records in format $(x, F_i(x), k)$. It denotes that the hint frequency of block x in period from kT to $(k+1)T$ is $F_i(x)$. T_{out} retrieves entries to compute hotness value only when a block is inserted into a cache level. Simultaneously, it inserts or updates records only when a block is leaving a cache level. For example, block x_1 is evicted from a cache level at time t and $k_1T < t < (k_1+1)T$. Then a new record $(x_1, F_i(x_1), k_1)$ is inserted into T_{out} . If there has already existed a record $(x_1, F'(x_1), k_1)$ in T_{out} , the record will be replaced by $(x_1, F(x_1), k_1)$. The old records (resided longer than kT) in T_{out} are marked as invalid. When T_{out} is full, a cleanup process is called to remove all the invalid records. If T_{out} still has no space for the new record insertion after the cleanup process, a record with the lowest hint frequency in those oldest time intervals is evicted.

The HFA management policy is shown in Algorithm 1. Before kickoff, HFA sets a constant time interval T , a degradation ratio δ and creates two queues Q_1 and Q_2 on each cache level (line 1). For cache misses,

Algorithm 1. Pseudo-Code for HFA Cache Operations

Input: a data block x accessed on cache level i

Initialize:

1: Set T and δ , and create Q_1 and Q_2

Cache miss:

2: **if** x can be found in T_{out} **then**

3: Calculate $F_i(x)$

4: **if** $F_i(x) > F_{min}$ **then**

5: Call $InsertQ_1(x, F_i(x))$

6: **else**

7: Insert x into Q_2

8: **end if**

9: **else**

10: Insert x into Q_2

11: **end if**

Cache hit in Q_1 :

12: $Hit(x)++$

13: Compare blocks ahead of x until it finds a block x' which satisfies $H(x') > H(x)$

14: Insert x behind x'

Func $InsertQ_1(x, F_i(x))$

Input: a data block x and $F_i(x)$

15: Set $H_i(x) = F_i(x)$

16: **if** Q_1 is full **then**

17: Evict block x' with the minimal hotness from Q_1 to Q_2

18: **end if**

19: Insert x to Q_1

20: **if** $hits(x') > hits_{min}$ **then**

21: Insert x' to Q_2

22: **else**

23: Call $Demote(x')$ to demote x' to a lower level

24: **end if**

HFA firstly checks whether corresponding records of a missing block x can be found in T_{out} . If so, it reads T_{out} entries to calculate the hint frequency $F_i(x)$ and decides which queue the block x should reside (lines 2 and 3). If $F_i(x)$ is larger than F_{min} , HFA calls a function `InsertQ1` to insert x in Q_1 . Otherwise, x is put in Q_2 . F_{min} denotes a threshold value which is the minimal hint frequency that blocks need to meet to stay in Q_1 (lines 4~8). In our implementation $F_{\text{min}} = 1$ and $k = 6$. If x has no records in T_{out} , HFA sends x to the head of Q_2 (line 10).

The function `InsertQ1` takes a data block and its hint frequency value as inputs. If Q_1 is full, it evicts block x' with the minimal hotness from the queue, and then inserts x into Q_1 (lines 16~19). The block x' is either demoted to a lower level or inserted into Q_2 , by comparing $hits(x')$ with the lowest cache hits in Q_2 , which is denoted by $hits_{\text{min}}$. If $hits(x')$ is larger, then x' is sent to Q_2 , otherwise, it calls function `Demote` to demote x' to a lower level (lines 20~24). The demote policy will be introduced in Subsection 3.3. It is worth noting that HFA is an exclusive multilevel cache algorithm, and it never discards data blocks. The evicted block from one queue is either demoted or moved to another queue.

3.2.3 Lazy Update and Bubble Sort

The hint frequency of a block is a dynamic value, which is degrading along with time. Therefore, sorting blocks by their hotness values in Q_1 requires a real-time update which is costly and impractical for cache management. Therefore, we propose a lazy update and bubble sort to solve this problem.

Firstly, regarding the lazy update, we set a hotness expire duration (T) and a last update time for all blocks. We say the hotness of a block is expired if the duration from the last update time is longer than T . And we check T_{out} to update $H_i(x)$ when it is expired and x is called to use.

The demote and the promote processes of HFA work on the blocks with the minimal and the maximal hotness values, respectively. However, selecting those data blocks needs a sorting through all the blocks in a queue. To avoid this scenario, we introduce a bubble sort mechanism. The bubble sort increases $hits(x)$ by 1 if there is a cache hit on block x , and then it compares $H(x)$ with the blocks ahead sequentially until it finds block x' which satisfies $H(x') > H(x)$. Then, block x is allocated behind x' (lines 12~14 in Algorithm 1). Instead of searching the block with the maximal and the minimal hotness values, the bubble sort only needs to

retrieve the head and the tail blocks in a queue. Therefore, the bubble sort improves the sorting performance without correctness garranteed.

3.3 Demotion/Promotion Policies Among Multiple Levels

According to the cache policies in each level and history hint information, we adjust the original demotion/promotion policies for demote/promote-based cache algorithms, such as `Demote`^[12], `Promote`^[13], and `Hint-K`^[22-23]. HFA uses hint frequency value as a hint. The hint is bundled with the corresponding data block and demoted or promoted to the other cache levels.

The demotions and promotions can happen on either Q_1 or Q_2 . Since Q_2 is managed separately by a proposed multilevel cache policy, we only demonstrate the demotion and promotion process for Q_1 . As shown in Algorithm 2, the demotion policy is implemented by function `Demote`, which takes a block x with the minimal hotness in Q_1 as an input. Before the demotion of x from cache level $i - 1$, HFA updates T_{out} with $(x, F_i(x), k)$, and takes $F_i(x)$ as a hint. Then, $F_i(x)$ and block x are sent to level i . When cache level i receives x , HFA takes the hint and history records in T_{out} to calculate an initial hotness value $H_i(x)$ for the demoted block x by using (3). Then, $H_i(x)$ decides into which queue the block x should be inserted.

Algorithm 2. Pseudo-Code for HFA Demotion and Promotion Processes

Func Demote(x)

Input: a data block x with the minimal hotness of Q_1

In cache level $i - 1$

- 1: Insert $F_i(x)$ into T_{out}
- 2: Send $F_i(x)$ and block x to a lower level i
- 3: Evict x from Q_1

In cache level i

- 4: Calculate $H_i(x)$
- 5: Insert x into a queue
- 6: **if** x is inserted into Q_1 **then**
- 7: Call `Promote`(x)
- 8: **end if**

Func Promote(x)

Input: a data block x added into Q_1

In cache level i

- 9: Pick up a block x' from the head of Q_1
 - 10: **if** $H_i(x') > H_i(x)$ **then**
 - 11: Insert $F(x')$ into T_{out}
 - 12: Send $F(x')$ and block x' to an upper level $i - 1$
 - 13: Evict x' from Q_1
 - 14: **end if**
-

Considering that the blocks with higher hint frequencies tend to be reused in a near future (shown in

Fig.1), HFA promotes hot blocks in Q_1 from a lower cache level i to a higher level $i - 1$ to improve the cache response time on the hot blocks. The promotion process is awakened in a cache level i when Q_1 receives a demoted block x from $i - 1$. Assuming that x' is a block with the maximal hotness in Q_1 of level i . HFA compares $H_i(x')$ with $H_i(x)$, and promotes x' to level $i - 1$ if $H_i(x')$ is larger. After promotion, $F(x')$ is recorded in T_{out} and x' is evicted from Q_1 .

3.4 Dynamic Partition

Dual-queue design in HFA prolongs the life of blocks with higher hint frequency in a cache level by managing the hot blocks in Q_1 . The hot blocks in Q_1 are more likely to be visited again in a near future. However, an improper configuration on queue size impacts the cache performance negatively. A larger Q_1 could result in wasting the cache space for keeping cold blocks in Q_1 , but replacing blocks in Q_2 more frequently. On the contrary, a smaller Q_1 loses the benefits gained from the hint frequency properties.

HFA uses a dynamic partition strategy in runtime to adjust the size ratio between Q_1 and Q_2 . The basic idea of the dynamic partition is to enlarge Q_1 (shrink Q_2) if it contains a number of cold blocks ($H_i(x) = 0$ is considered as cold blocks in our configuration), but to shrink Q_1 (enlarge Q_2) if Q_2 contains too many hot blocks. The exchanged queue space between Q_1 and Q_2 is estimated by equation $S = N_C - N_H$, where N_C and N_H are the number of cold and hot blocks in Q_1 and Q_2 respectively.

In our implementation (see Algorithm 3), HFA repartitions the two queues over every T time in each cache level. Initially, the whole cache space is allocated to Q_1 and Q_2 evenly. Next, in run-time, HFA records the resident duration of cold blocks $t_C(x)$ in Q_1 and hot blocks $t_H(x)$ in Q_2 for every period T . And then it calculates S by using the average number of cold and hot blocks in Q_1 and Q_2 respectively.

$$S = \overline{N_C} - \overline{N_H},$$

where $\overline{N_C} = \frac{\sum_{x \in Q_1} t_C(x)}{T}$ and $\overline{N_H} = \frac{\sum_{x \in Q_2} t_H(x)}{T}$. The value will round down to the nearest integer. If S is a positive value, it indicates that the negative impact of cold blocks in Q_1 outweighs that of hot blocks in Q_2 . Thereby, HFA resizes the two queues by moving S space from Q_1 to Q_2 . If Q_1 is full, S blocks are also reallocated to Q_2 from the tail of Q_1 , vice versa. Note that the partitioning is logical, not physical, and the

cache blocks are not actually moved in the cache during repartition.

Algorithm 3. Dynamic Partition Process of HFA

```

Func Partition()
For each cache level
  Cache space is allocated to  $Q_1$  and  $Q_2$  evenly
  for each  $T$  time do
    Calculate  $S$ 
    if  $S > 0$  then
      Move  $S$  space from  $Q_1$  to  $Q_2$ 
    end if
    if  $S < 0$  then
      Move  $S$  space from  $Q_2$  to  $Q_1$ 
    end if
  end for

```

4 Case Study

HFA uses hint frequency information to improve the cache hit ratio in each cache level. Although the hint frequency predicts the cache access property in each level, it has two defects. Firstly, the hint frequency is a collection of history demotion and promotion operations, which usually has latency. For recency- and frequency-dominated applications, it may not catch the cache access pattern accurately. Secondly, the hint frequency improves the cache hit ratio of the blocks departure or arrival frequently in a period on each cache level. It does not catch the blocks that travel actively in different cache levels. For example, in a period T , a block x moves in and out level i for m times, where $m > F_{min}$. It is considered as a hot block and will be kept longer in Q_1 in level i . Nevertheless, if x travels several different cache levels after it leaves level i , it will not be marked as a hot block. As proposed in [23], these blocks are considered as active blocks and have a potential to impact cache performance for many workloads.

The two-queue design of HFA makes it easy for cooperating with the other cache approaches to further improve the cache performance. In this section, we show how HFA integrates with algorithms like Demote, Promote and Hint-K to manage multilevel cache.

4.1 HFA Combined with Demote

Demote policy^[12] introduces a DEMOTE operation to achieve exclusive caching. The blocks ejected from an upper cache are transferred to a lower level. Demote policy improves the cache system response time while the operation overhead is comparatively lower to the other multilevel cache approaches. Regarding

to the latency issue of HFA, HFA uses demote policy to manage Q_1 and sort blocks in Q_2 by hotness separately as shown in Algorithm 4. As a result, the MRU data blocks are marked as a hot block without waiting for any Demote/Promote operations. Thereby, Demote can reduce the latency for the recency-dominated applications.

Algorithm 4. Cache Management for HFA Combined with Demote Policy

For a cache read miss in cache level i :

- 1: Get missing block x from a lower level cache or disk
- 2: Compute $F_i(x)$, and put x into Q_2
- 3: If Q_2 is full, then demote a block from the queue tail to level $i + 1$
- 4: Update T_{out} with the demoted block

For a cache hit in Q_2 or Q_1 :

- 5: Put the hit block to the head of Q_1
 - 6: **if** Q_1 is full **then**
 - 7: Demote the tail block to level $i + 1$, and update T_{out}
 - 8: **end if**
-

4.2 HFA Combined with Promote

Promote approach^[13] applies PROMOTE operation instead of DEMOTE. It achieves exclusive caching without demotions while the bandwidth cost and the cache aggregate response time are lower than those of DEMOTE. The principle of PROMOTE is to push the blocks that are most likely to be visited to the upper level. Combining HFA with PROMOTE can actively increase the hotness of those promoted blocks, which reduces the delay for hot blocks identification. To adapt PROMOTE, HFA assigns one bit hint for each block $hintP(x)$ to decide which cache should own the block. Computation of $hintP(x)$ in our implementation is the same with [13]. $hintP(x)$ is set to true when a cache hit or disk read is firstly formed and set to false when some cache decides to keep it. The management policy is shown in Algorithm 5.

4.3 HFA Combined with Hint-K

Hint-K approach explores the relationship between cache access and block activeness which is the degree of block movements between cache levels. Regarding to the second issue of HFA, Hint-K can gather the block history information from different cache levels which improves the hit ratio of HFA. Combining HFA and Hint-K needs some changes on the cache replacement policy (as shown in Algorithm 6). Instead of demotion and promotion with hint frequency, the combined algo-

rithm uses K-step Hint Value (KHV)^[23] as a hint. Q_2 is sort by LRU (Last Recent Use).

Algorithm 5. Cache Management for HFA Combined with Promote Policy

For a read request from upper level $i - 1$

- ```

if cache miss in Q_2 then
 Send request to level $i + 1$
else
 Calculate $hintP(x)$
 if $hintP(x)$ is True then
 Update T_{out} and promote x to level $i - 1$
 end if
end if

```

**For receiving a block  $x$  from the other levels**

- ```

if  $F_i(x) \leq F_{min}$  then
  if  $hintP(x)$  is True then
    Update  $T_{out}$  and promote  $x$  to level  $i - 1$ 
  else
    Insert  $x$  in  $Q_2$ 
  end if
else
  Insert  $x$  in  $Q_1$ 
end if

```
-

Algorithm 6. Cache Management for HFA Combined with Hint-K Policy

**For Q_2 of level i ,
receiving a block x :**

- ```

if $x \in T_{out}$ then
 Insert x into Q_1
else
 Insert x into the head of Q_2
end if

```

**demoting a block:**

- Step 1.** Get the minimum KHV in  $Q_2$
- Step 2.** Find block(s) with the minimal KHV
- Step 3.** Demote the last block from those in step 2 to level  $i + 1$
- Step 4.** Update KHV value of the demoted block

**promoting a block at read miss:**

- Step 1.** Receive the minimum KHV in  $Q_2$  from level  $i - 1$
  - Step 2.** If KHV of  $x$  is larger than the minimum KHV, then promote  $x$  and update corresponding KHV
- 

Hint-K<sup>[22-23]</sup> is the closest approach to our work. Both approaches use a hybrid hint to improve the multi-level cache performance. However, there are two major differences between Hint-K and HFA. 1) The management policy of Hint-K in each cache level does not consider the period/sudden hot blocks, which means a block becomes a hotspot data in a short period. For example, for block  $x$  that was hopped between cache levels frequently in a transient time interval, Hint-K considers it as an active block and assigns  $x$  a high KHV to capture  $x$  in an upper level. If  $x$  is never used in the near future, we prefer to remove  $x$  from the queue im-

mediately. On the contrary, Hint-K resists  $x$  due to its high KHV, which reduces the cache hit ratio. This issue can get worse when those cold blocks are accumulated. Different from Hint-K, HFA considers this issue by degrading hint over time. A block can get cold and be demoted to a lower level when it receives no operation in a period. 2) Hint-K made an observation that hot blocks are usually active. In another word, a block is identified to be hot only after a number of inter-cache level movements of the block. Hence, there are always a latency and extra network traffic (typically, they are called warm-up cost of a data block<sup>[53]</sup>) to capture hot blocks in Hint-K. Instead, HFA reduces the warm-up response to hot blocks. We define a hot block by both hit and movement frequencies, which guarantees a quick detection on hot blocks and holds it in a higher priority queue.

## 5 Simulation Methodology and Analysis

To demonstrate the effectiveness of the HFA algorithm, we use a trace-driven simulation to evaluate HFA and other popular multilevel cache approaches under different I/O workloads.

### 5.1 Simulation Methodology

We use *fschasesim* as the simulator to evaluate various multilevel cache solutions, which appears in several previous literatures<sup>[12,22-23]</sup>. Nowadays most multilevel cache algorithms are based on demote and promote hints<sup>[12,16,22,42-44]</sup>, thereby we select Demote<sup>[12]</sup>, Promote<sup>[13]</sup> and Hint-K<sup>[22-23]</sup> algorithms in our comparison. HFA approach is collaborated with these algorithms, and we use D-HFA, P-HFA, H-HFA to delegate HFA combined with demote hints, promote hints and hybrid hints (both demote and promote hints), respectively. We select an ideal case on the knowledge of future hint information called Oracle HFA (Oracle for short), which is included in our comparison as well.

We use six I/O traces in our simulation as below. Statistics of the six traces are summarized in Table 3.

1) *WebSearch*: this I/O trace is collected from a popular search engine.

2) *FIU IODedup Homes (FIU)*: the block traces are the collected downstream of an active page cache for three weeks from an NFS server that serves the home directories of SNIA research group.

3) *MSN Storage File System (MSN-SFS)*: the traces are collected for MSN storage file server for a duration of 6 hours, which consists of 36 10-minute trace files;

they trace the primarily disk I/O events at block level as well as file I/O events.

4) *Microsoft Exchange*: the Microsoft Exchange traces are collected from an Exchange 2007 SP1 server, which is a mail server for 5000 corporate users.

5) *TPC-E*: the traces are collected at Microsoft running TPC-C benchmark.

6) *TPC-C*: the traces are collected at a server running TPC-C benchmark.

**Table 3.** Summary of Traces

| Trace              | Year | Size of Dataset (GB) | Length (h) | I/Os ( $\times 10^6$ ) |
|--------------------|------|----------------------|------------|------------------------|
| WebSearch          | 2002 | 4.3                  | 2.6        | 5.7                    |
| FIU                | 2009 | 5.9                  | 5.7        | 7.3                    |
| MSN-SFS            | 2007 | 1.6                  | 6.0        | 4.5                    |
| Microsoft Exchange | 2007 | 1.9                  | 5.0        | 11.5                   |
| TPC-E              | 2007 | 18.4                 | 11.4       | 16.7                   |
| TPC-C              | 2007 | 27.2                 | 12.1       | 21.3                   |

Unless otherwise mentioned, the following default parameters are used: two cache levels ( $n = 2$ ) and the block size (4 KB). The ratio of cache size between an upper cache level and the next lower level is 1:4, and it is 1:2 for the Websearch and FIU traces. Aggregate cache size is the sum of all cache levels. Based on the default settings of *fschasesim*, the average access time of  $L_1$  cache,  $L_2$  cache and disks is 0.2 ms, 2 ms and 10 ms, respectively. We set the warm-up time to be long enough to make sure enough data blocks have been flooded to all cache levels.

In our simulation, we set proper values for several parameters in HFA. The size of  $T_{out}$  table is set to 0.1% of the cache size of each level, and  $\delta$  is set to 0.5.  $T$  is set to the duration of 1000 requests. We find that this group of setting gives the best performance in the experiment.

### 5.2 Simulation Results

In this subsection, we give the simulation results of different multilevel cache algorithms under various I/O workloads.

#### 5.2.1 Cache Performance

First, we measure the cache performance under different workloads. In these simulations, we use two typical metrics, aggregate hit ratio and average response time, to evaluate the I/O performance.

The results of aggregate hit ratio are presented in Fig.4. It is clear that compared with the corresponding original cache algorithms such as Demote, Promote

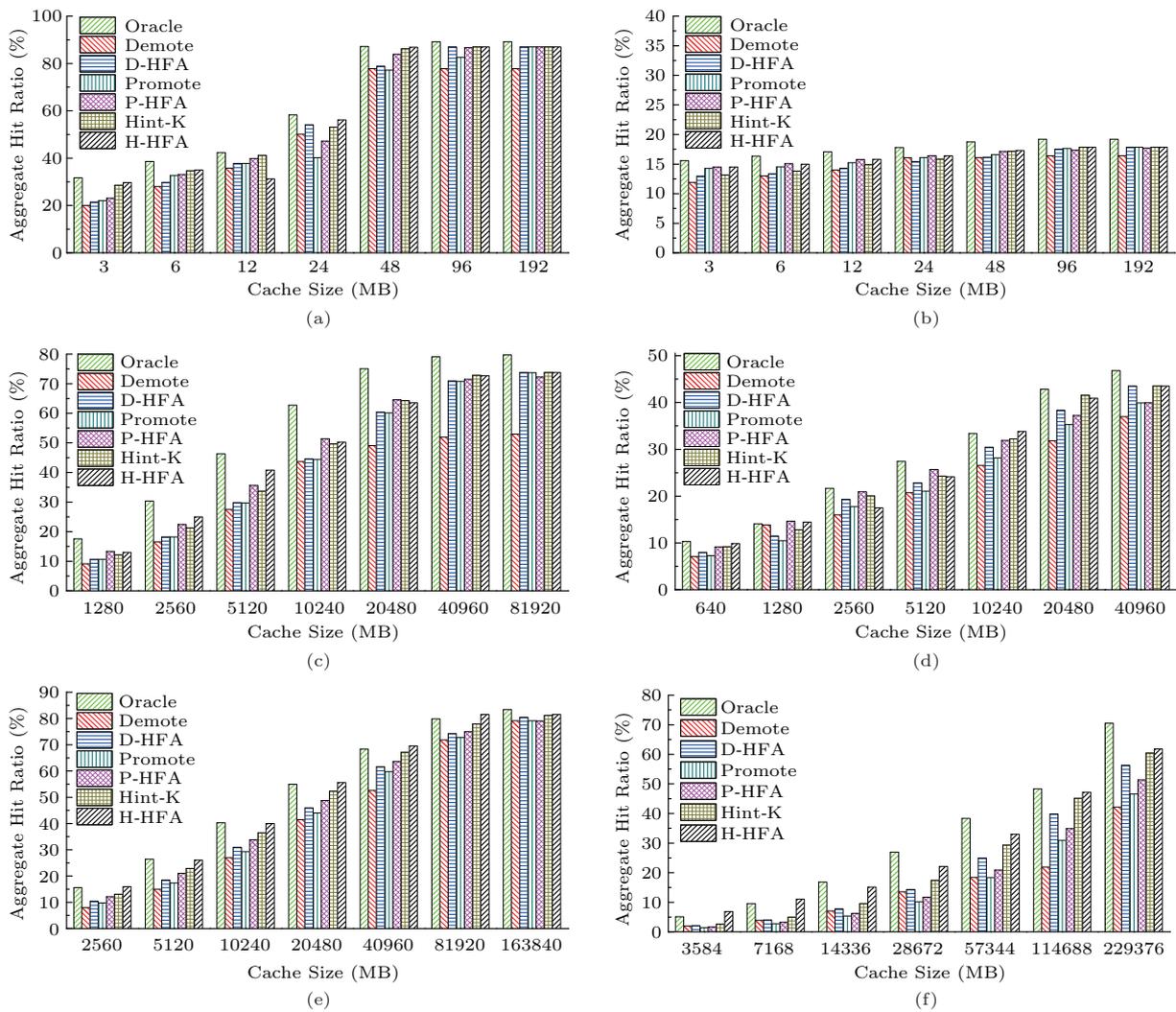


Fig.4. Aggregate hit ratio under various traces with different aggregate cache sizes. FIU-HOMES uses a two-level cache hierarchy which has a fixed cache size ratio of 1:2. MSN-SFS, Microsoft Exchange and TPC-E use a two-level cache hierarchy with a fixed cache size ratio of 1:4. TPC-C uses a three-level cache hierarchy with a fixed cache size ratio of 1:2:4. (a) WebSearch. (b) FIU-HOMES. (c) MSN-SFS. (d) Microsoft Exchange. (e) TPC-E. (f) TPC-C.

and Hint-K, HFA achieves performance gains by up to 19.1%, 7.1% and 6.5%, respectively. In most cases, it is easy to see that the gains from HFA are proportional to the cache size. This is because large cache size is suitable for HFA to allocate blocks to right queues. Further, increasing cache size beyond a value (such as 48 MB of WebSearch and 40 906 MB of MSN-SFS) does not seem to help hit ratio significantly. Moreover, compared with Hint-K, HFA presents a little improvement on Hint-K for various workloads, as shown in Figs.4(a)~4(d). The reason is that those workloads show strong temporal locality. Then, in runtime, the size of  $Q_1$  is much larger than the size of  $Q_2$ . As a result, the behaviour of H-HFA approaches to that of Hint-K.

The simulation results on response time under different workloads are shown in Fig.5. Compared with original Demote, Promote and Hint-K algorithms, HFA sharply decreases the average response time. For example, D-HFA reduces the average response time by up to 26% compared with Demote in Fig.5(c). P-HFA and H-HFA decrease the I/O latencies by up to 20.7% and 10.1%, respectively. Clearly, the latency improvement of HFA over Hint-K is comparatively smaller than that of D-HFA and P-HFA. It is because that both HFA and Hint-K are history hint based approaches. In Fig.5(b), we also notice a special case on cache size 96 MB that Promote responses faster than HFA. The reason is that, for trace FIU-HOMES, a large number of promoted blocks managed by HFA in the upper level are degraded

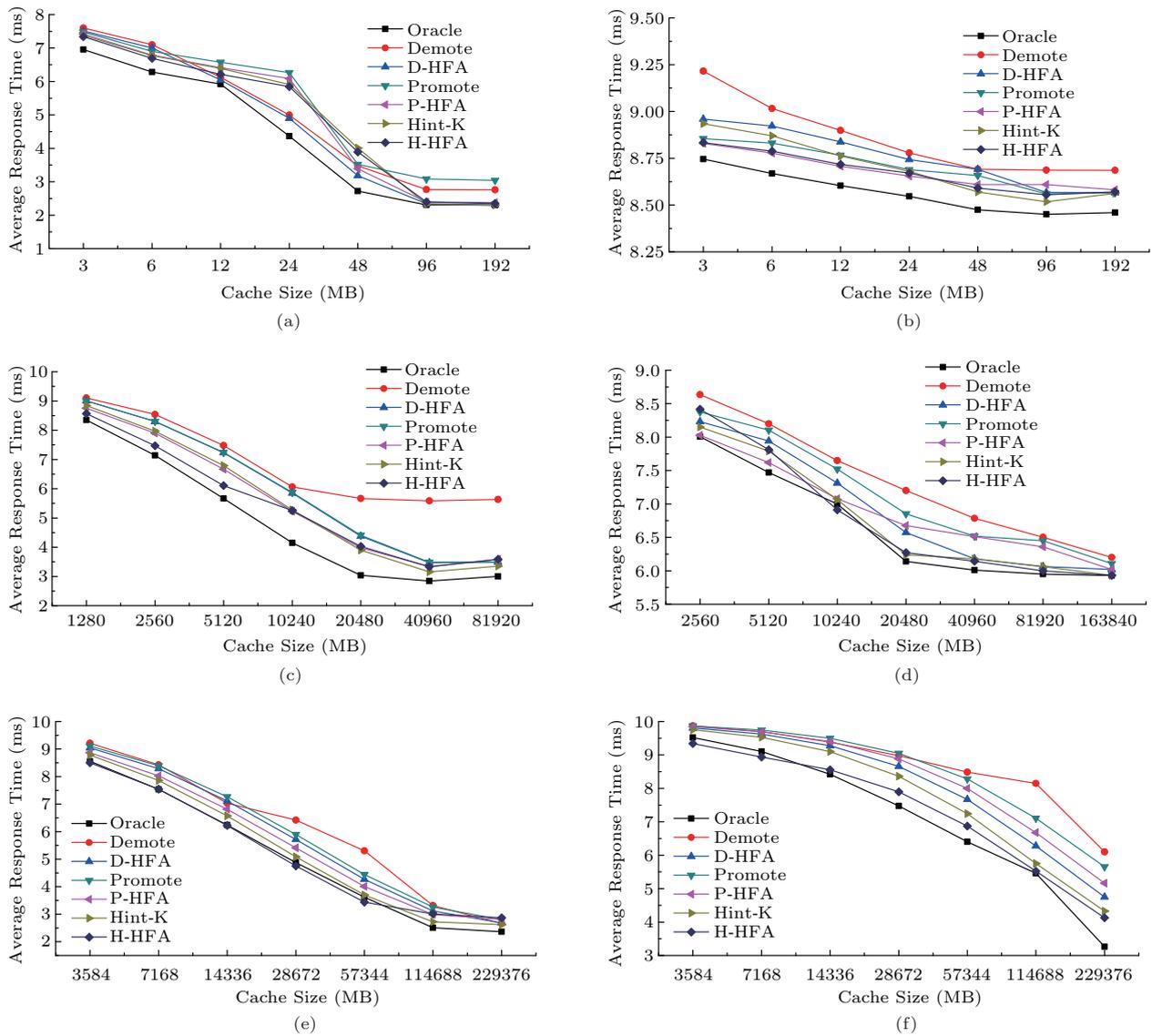


Fig.5. Average response time under different traces with different aggregate cache sizes. The TPC-C trace runs at a three-level cache architecture, and the cache size ratio is 1:2:4. (a) WebSearch. (b) FIU-HOMES. (c) MSN-SFS. (d) Microsoft Exchange. (e) TPC-E. (f) TPC-C.

as cold blocks and demoted quickly before the next visiting.

We summarize the previous results as shown in Table 4.

**Table 4.** Improvement of HFA Approach over Other Typical Multilevel Cache Algorithms in Terms of Aggregate Hit Ratio and Average Response Time

| Algorithm | Aggregate Hit Ratio (%) | Average Response Time (%) |
|-----------|-------------------------|---------------------------|
| Demote    | 19.1                    | 22.0                      |
| Promote   | 7.1                     | 20.1                      |
| Hint-K    | 6.5                     | 10.1                      |

### 5.2.2 Cache Overhead

The first overhead in our evaluation is the consumption of inter-cache bandwidth of different algorithms, and the results are shown in Fig.6. The number of hint frequencies of all data blocks under different workloads is monitored. Obviously, HFA reduces hint frequencies by approximately 10% on average. It demonstrates that HFA is more efficient on keeping the hot data blocks at a proper level. We also notice that the enhancement becomes larger with the increased cache size.

Regarding to the HFA scheme, the space overhead is decided by  $T_{out}$  and  $Q_1$ . In our simulation, the size of  $T_{out}$  is fixed to 0.1% of the corresponding cache size

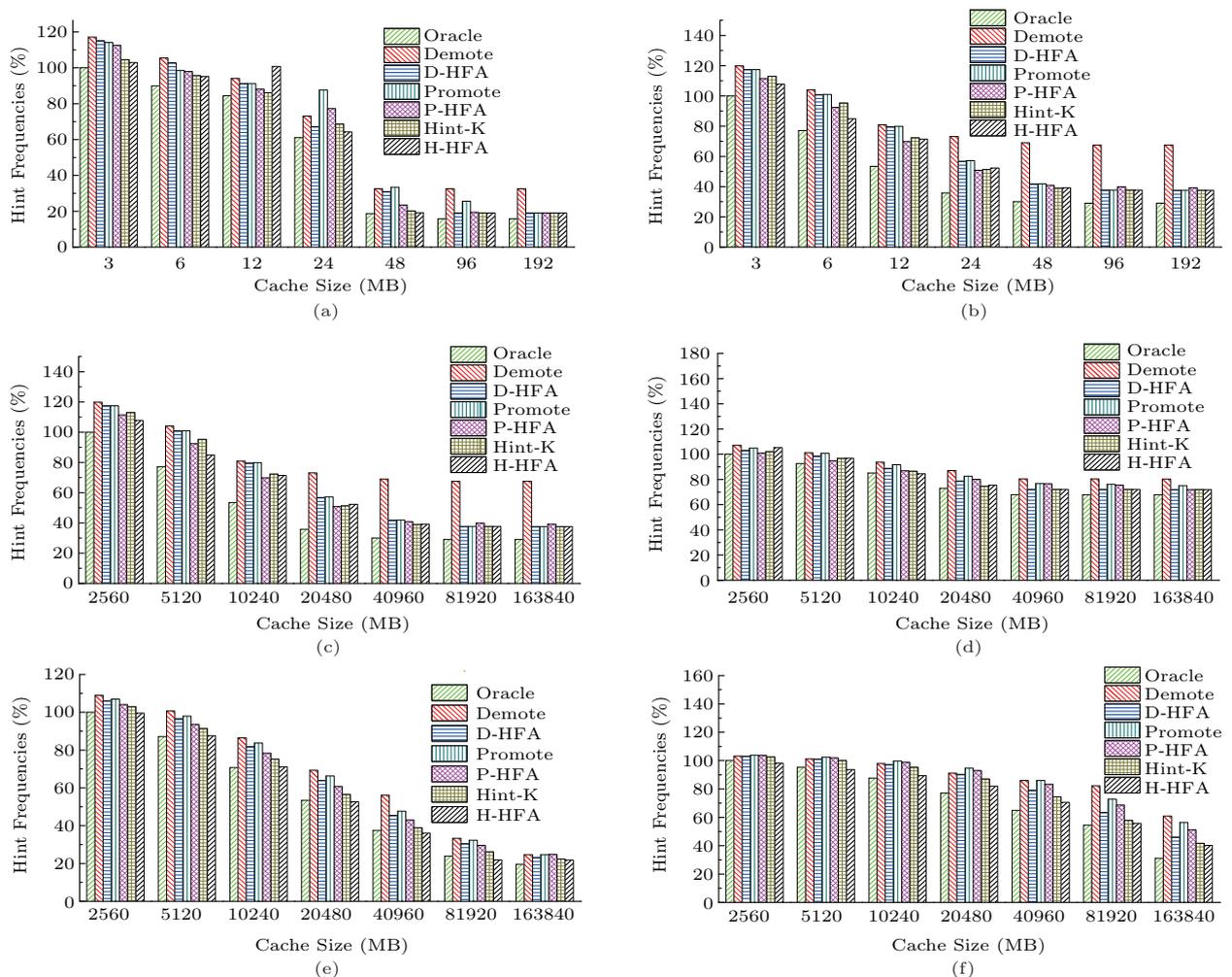


Fig.6. Hint frequencies of all data blocks under different I/O workloads (the hint frequencies in Oracle HFA are normalized to 100%). (a) WebSearch. (b) FIU-HOMES. (c) MSN-SFS. (d) Microsoft Exchange. (e) TPC-E. (f) TPC-C.

in each level.  $Q_1$  uses extra space (a few megabytes in typical) to record the history hint information of hot data blocks.

The space overhead is shown in Fig.7. It is clear that the space overhead of HFA is very low (less than 2.6%).

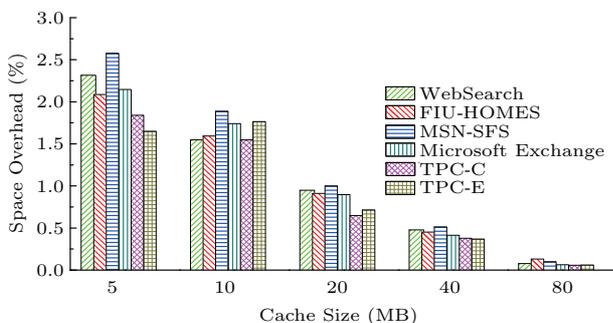


Fig.7. Space cost of HFA approach under different traces.

### 5.2.3 Stability

The stability of a cache system shows the changes on the system response speed in run time and how quick the system can reach a stable hit ratio in the first cache level after start-up. In this part we analyze the stability of HFA by choosing TPC-C as the benchmark, and implementing D-HFA, P-HFA and H-HFA on a three-level cache with the aggregate cache space of 114 GB. The cache space ratio of the three levels are 1 : 2 : 4.

In Fig.8, the  $x$ -axis is the time that benchmarks run for 512 periods from the time when all the cache levels are warmed up. The  $y$ -axis of Fig.8(a) and Fig.8(b) is the cache hit ratio and the space proportion of  $Q_1 : Q_2$ , respectively. It is noted that the proportion in the first time period ( $T = 1$ ) is 1 : 1 for all three approaches.

Fig.8(a) shows the first level cache hit ratio for the three approaches along with the time. It is clear that

the hit ratio reaches a stable level at around 54, 39 and 62 after 8, 6 and 4 periods for D-HFA, P-HFA and H-HFA, respectively. The fluctuation of H-HFA (8.9% on average) is larger than the fluctuation of D-HFA (4.6%) and P-HFA (4.8%).

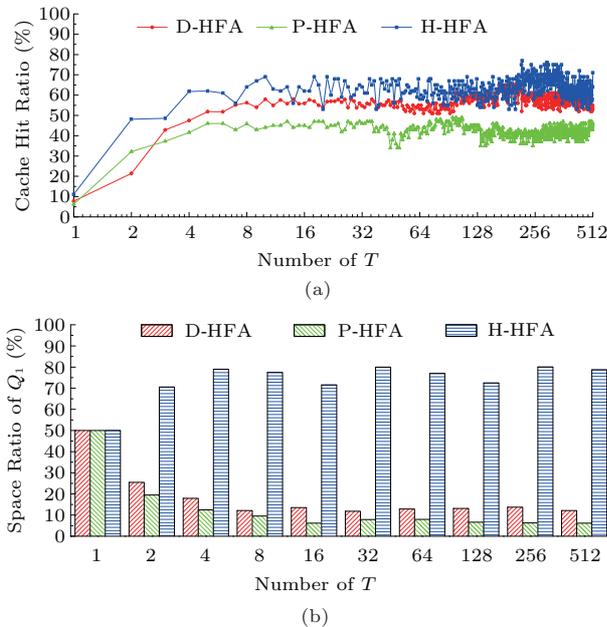


Fig.8. Influence of dynamic partition on stability over time. (a) Comparison of cache hit ratio on the first level under different approaches. (b) Comparison of queue proportion under different approaches.

Fig.8(b) illustrates the results from the dynamic partition of changing of proportion between  $Q_1$  and  $Q_2$  along with the time. For D-HFA and P-HFA,  $Q_1$  takes 13% and 9% of the whole cache space of the top level, respectively. On the contrary,  $Q_1$  takes over 70% of the cache space in H-HFA. This is because that Hint-K assists to identify hot blocks from lower levels and aggregates them to the top level. Then the refresh frequency of blocks in the first cache level increases. Consequently,  $Q_1$  requests more space to keep the hot blocks. Meanwhile, this is also the reason why the fluctuation of H-HFA is larger than that of the other approaches.

#### 5.2.4 Impact of Size Ratio of Two Queues

Fig.9 examines the impact of size ratio of  $Q_1$  and  $Q_2$  on the cache aggregate hit ratio. We use TPC-C as the benchmark in this evaluation. The aggregate cache size is 114 GB. The cache space ratio of the three levels is 1 : 2 : 4. Instead of dynamic partition, the space proportion of  $Q_1 : Q_2$  is increased manually from 0% to 100% with an interval of 10%. From the figure, we

notice that the aggregate hit ratio of D-HFA, P-HFA and H-HFA reaches a peak value of 46%, 35% and 55% at a space ratio of 2 : 8, 1 : 9, and 8 : 2, respectively, which is about 11% lower than the value that uses dynamic partition. It can be seen that P-HFA uses the smallest  $Q_1$  space than the rest to reach the maximal aggregate hit ratio, because TPC-C has a weak temporal locality. Most of the blocks promoted from the lower level are stored in  $Q_2$  and usually missed. This is another reason why D-HFA performs better on TPC-C. Moreover, it can be seen that  $Q_1$  of H-HFA consumes over 80% of the whole cache space to meet the best hit ratio. As illustrated in Subsection 5.2.3, Hint-K increases the refresh frequency of blocks, which increases the hotness value of the block as well. As a result, it allocates more cache space to  $Q_1$  to keep the hot blocks.

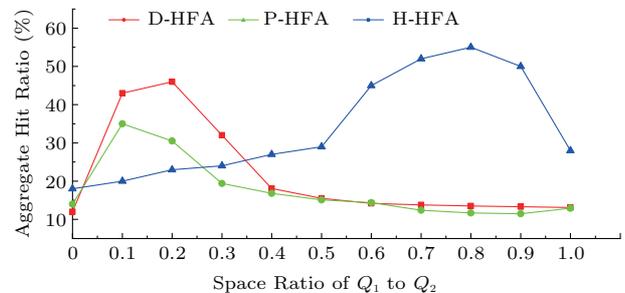


Fig.9. Aggregate hit ratio of D-HFA, P-HFA and H-HFA when varying the proportion between  $Q_1$  and  $Q_2$  on a three-level cache over trace TPC-C.

#### 5.2.5 Examination of Parameter $k$ of HFA

We now study the impact of parameter  $k$ . The trace and the cache configuration are the same with the previous evaluation. In Fig.10, we plot the parameter  $k$  versus the aggregate cache hit ratio. When  $k$  is zero,  $T_{out}$  is disabled, and the hotness of blocks is only related to the hit frequency which makes D-HFA, P-HFA and H-HFA behave close to original Demote, Promote and Hint-K respectively. It can be seen that the aggregate hit ratio increases to a peak value when  $k$  is 4 for P-HFA and 5 for both D-HFA and H-HFA, and then goes smoothly. The larger the value of  $k$ , the longer the operation history of blocks recorded in  $T_{out}$ , which calculates the hotness values more accurately. However, according to (3), the old history (large  $k$ ) contributes much less to the hotness than new operations. Further, since  $T_{out}$ 's size is fixed, the old history could be removed by the clean-up process. As a result, after parameter  $k$  grows to a proper value, the aggregate hit ratio will not increase anymore. Therefore, in the configuration of HFA, we set  $k$  to a fixed value of 6 for all workloads.

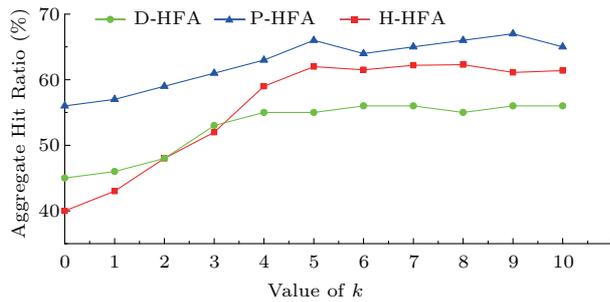


Fig.10. Plot of the parameter  $k$  (the steps of history operations) versus the aggregate hit ratio for algorithms D-HFA, P-HFA and H-HFA on trace TPC-C. The cache has three levels and the space ratio of each level is 1:2:4. The aggregate cache space is 114 GB.

### 5.3 Analysis

From the results in Subsection 5.2, compared with the original Demote, Promote and Hint-K algorithms, it is clear that HFA approach has many advantages on cache performance. There are several reasons to achieve these gains. First, HFA is based on the analysis of the history hint information of data blocks, which is one of the most significant essences in multilevel cache systems. By effectively identifying the hint frequencies, hot data blocks are selected to have a long residence in upper-level cache, which increases the aggregate hit ratio of data blocks and decreases the average response time. Second, HFA divides traditional queues into two dedicated queues, which keeps the hint information locally to reduce the usage of inter-cache bandwidth. Third, HFA reduces the overall hint frequencies (inter-cache demotions/promotions), which decreases the bandwidth consumption among different levels. Besides, the space cost of HFA approach is very low, which can help to provide high performance under various workloads. In addition, HFA exhibits a good stability from two aspects. On one hand, the turbulence on the cache hit ratio is low. On the other hand, it can reach a stable status in a short time.

## 6 Conclusions

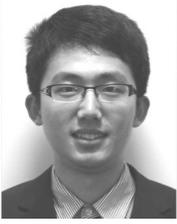
In this paper, we proposed a novel multilevel cache approach called HFA, which explores hint frequencies of data blocks. HFA monitors the status of data blocks to keep hot data blocks with high hint frequencies resident in cache as long as possible. Mathematical definitions are given to effectively identify hot data blocks. And we discussed various types of hints combined with the HFA approach. The simulation results showed that, compared with original Demote, Promote and Hint-K algorithms, the corresponding algorithms combined

with HFA approach achieve better performance under different I/O workloads.

## References

- [1] Zhao Q, Liew S C, Zhang S, Yu Y. Distance-based location management utilizing initial position for mobile communication networks. *IEEE Transactions on Mobile Computing*, 2016, 15(1): 107-120.
- [2] Yang R, Wang Z. Cross-oriented choquet integrals and their applications on data classification. *Journal of Intelligent & Fuzzy System*, 2015, 28(1): 205-216.
- [3] Zhu Z, Zhang Y, Ji Z, He S, Yang X. High-throughput DNA sequence data compression. *Briefings in Bioinformatics*, 2015, 16(1): 1-15.
- [4] Wang T, Liu D, Wang Y, Shao Z. Towards write-activity-aware page table management for non-volatile main memories. *ACM Transactions on Embedded Computing Systems*, 2015, 14(2): 34:1-34:23.
- [5] Liu J K, Au M H, Susilo W, Liang K, Lu R, Srinivasan B. Secure sharing and searching for real-time video data in mobile cloud. *IEEE Network*, 2015, 29(2): 46-50.
- [6] Li J, Liu C, Liu B, Mao R, Wang Y, Chen S, Wang Q *et al.* Diversity-aware retrieval of medical records. *Computers in Industry*, 2015, 69: 81-91.
- [7] Huang X, Cheng H, Li R H, Qin L, Yu J X. Top-K structural diversity search in large networks. *The VLDB Journal*, 2015, 24(3): 319-343.
- [8] Chen R, Qin Z, Wang Y, Liu D, Shao Z, Guan Y. On-demand block-level address mapping in large-scale NAND flash storage systems. *IEEE Transactions on Computers*, 2015, 64(6): 1729-1741.
- [9] Hao Y, Xu J, Bai J, Han Y. Image decomposition combining a total variational filter and a Tikhonov quadratic filter. *Multidimensional Systems and Signal Processing*, 2015, 26(3): 739-751.
- [10] You Z H, Yu J Z, Zhu L, Li S, Wen Z K. A MapReduce based parallel SVM for large-scale predicting protein-protein interactions. *Neurocomputing*, 2014, 145: 37-43.
- [11] Motghare M, Shrawankar U. RFS-UCM: A unified multi-level cache management policy. In *Proc. the 9th IEEE Int. Conf. Intelligent Systems and Control*, Jan. 2015.
- [12] Wong T M, Wilkes J. My cache or yours? Making storage more exclusive. In *Proc. USENIX Annual Technical Conference*, June 2002, pp.161-175.
- [13] Gill B S. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *Proc. the 6th USENIX Conference on File and Storage Technologies*, Feb. 2008, pp.48-65.
- [14] Koltsidas I, Viglas S D. Designing a flash-aware two-level cache. In *Proc. the 15th Int. Conf. Advances in Databases and Information Systems*, Sept. 2011, pp.153-169.
- [15] Zhou Y, Philbin J, Li K. The multi-queue replacement algorithm for second level buffer caches. In *Proc. the USENIX Annual Technical Conference*, June 2001, pp.91-104.
- [16] Zhou Y, Chen Z, Li K. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 2004, 15(6): 505-519.
- [17] Appuswamy R, van Moolenbroek D C, Tanenbaum A S. Cache, cache everywhere, flushing all hits down the sink: On exclusivity in multilevel, hybrid caches. In *Proc. the 29th IEEE Symposium on Mass Storage Systems and Technologies*, May 2013.

- [18] Li X, Abounaga A, Salem K et al. Second-tier cache management using write hints. In *Proc. the USENIX FAST*, Dec. 2005.
- [19] Liu X, Abounaga A, Salem K, Li X. CLIC: CLient-informed caching for storage servers. In *Proc. the USENIX FAST*, Feb. 2009, pp.297-310.
- [20] Yadgar G, Factor M, Schuster A. Karma: Know-it-all replacement for a multilevel cache. In *Proc. the USENIX FAST*, Feb. 2007, pp.169-184.
- [21] Yadgar G, Factor M, Li K, Schuster A. MC<sup>2</sup>: Multiple clients on a multilevel cache. In *Proc. the 28th IEEE ICDCS*, June 2008, pp.722-730.
- [22] Wu C, He X, Cao Q, Xie C. Hint-*k*: An efficient multi-level cache using *k*-step hints. In *Proc. the 39th IEEE ICPP*, Sept. 2010, pp.624-633.
- [23] Wu C, He X, Cao Q, Xie C et al. Hint-K: An efficient multi-level cache using K-step Hints. *IEEE Transactions on Parallel and Distributed Systems*, 2014, 25(3): 653-662.
- [24] Denning P J. The working set model for program behavior. *Communications of the ACM*, 1968, 11(5): 323-333.
- [25] Robinson J T, Devarakonda M V. Data cache management using frequency based replacement. *ACM SIGMETRICS Performance Evaluation Review*, 1990, 18(1): 134-142.
- [26] Johnson T, Shasha D. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. the 20th Int. Very Large Data Bases*, Sept. 1994, pp.439-450.
- [27] 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.
- [28] O'neil E J, O'Neil P E, Weikum G. An optimality proof of the LRU-K page replacement algorithm. *Journal of the ACM*, 1999, 46(1): 92-112.
- [29] Kim J M, Choi J, Kim J et al. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proc. the 4th Symp. Operating System Design & Implementation*, Oct. 2000.
- [30] Lee D, Choi J, Kim J H et al. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Computers*, 2001, 50(12): 1352-1361.
- [31] 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.
- [32] Megiddo N, Modha D S. ARC: A self-tuning, low overhead replacement cache. In *Proc. the USENIX FAST*, Mar.31-Apr.2, 2003, pp.115-130.
- [33] Bansal S, Modha D S. CAR: Clock with adaptive replacement. In *Proc. the USENIX FAST*, Mar.31-Apr.2, 2004, pp.187-200.
- [34] Gniady C, Butt A R, Hu Y C. Program-counter-based pattern classification in buffer caching. In *Proc. OSDI*, Dec. 2004, pp.395-408.
- [35] Gill B S, Modha D S. SARC: Sequential prefetching in adaptive replacement cache. In *Proc. the USENIX Annual Technical Conference*, Apr. 2005, pp.293-308.
- [36] Zhou F, von Behren J R, Brewer E A. AMP: Program context specific buffer caching. In *Proc. the USENIX Annual Technical Conference*, Apr. 2005, pp.371-374.
- [37] Jiang S, Ding X, Chen F, Tan E, Zhang X. DULO: An effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proc. the 4th USENIX Conference on File and Storage Technologies*, Dec. 2005.
- [38] Jiang S, Chen F, Zhang X. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proc. the USENIX Annual Technical Conference*, Apr. 2005, pp.323-336.
- [39] Gill B S, Modha D S. WOW: Wise ordering for writes — Combining spatial and temporal locality in non-volatile caches. In *Proc. the 4th USENIX Conference on File and Storage Technologies*, Dec. 2005.
- [40] Zhu Y, Jiang H. RACE: A robust adaptive caching strategy for buffer cache. *IEEE Transactions on Computers*, 2008, 57(1): 25-40.
- [41] Gill B S, Ko M, Debnath B et al. STOW: A spatially and temporally optimized write caching algorithm. In *Proc. the USENIX Annual Technical Conference*, June 2009.
- [42] Bairavasundaram L N, Sivathanu M, Arpaci-Dusseau A C, Arpaci-Dusseau R H. X-ray: A non-invasive exclusive caching mechanism for raids. In *Proc. Annual International Symposium Computer Architecture*, June 2004, pp.176-187.
- [43] Chen Z, Zhou Y, Li K. Eviction-based cache placement for storage caches. In *Proc. the USENIX Annual Technical Conference*, June 2003, pp.269-281.
- [44] He X, Ou L, Kosa M J, Scott S L, Engelmann C. A unified multiple-level cache for high performance storage systems. *International Journal of High Performance Computing and Networking*, 2007, 5(1/2): 97-109.
- [45] Gill B S. Systems and methods for multi-level exclusive caching using hints. U.S. Patent US 7761664 B2, 2010.
- [46] Wang Y, Meng X, Zhang L et al. Chint: An effective and reliable cache management for RDMA-accelerated key-value stores. In *Proc. the ACM Symposium on Cloud Computing*, Nov. 2014.
- [47] Jiang S, Zhang X. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proc. the 24th Int. Conf. Distributed Computing Systems*, Mar. 2004, pp.168-177.
- [48] Yadgar G, Factor M, Li K et al. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Transactions on Computer Systems*, 2011, 29(2): 5.
- [49] Al Assaf M M, Alghamdi M I, Jiang X, Zhang J, Qin X. A pipelining approach to informed prefetching in distributed multi-level storage systems. In *Proc. the 11th IEEE Int. Symp. Network Computing and Applications*, Aug. 2012, pp.87-95.
- [50] Meng X, Zheng L, Li L et al. PAM: An efficient power-aware multi-level cache policy to reduce energy consumption of Software Defined Network. In *Proc. the 1st IEEE Int. Industrial Networks and Intelligent Systems*, Mar. 2015, pp.18-23.
- [51] Benhase M T, Gupta L M. Caching data in a storage system having multiple caches including non-volatile storage cache in a sequential access storage device. U.S. Patent US 8806122 B2, 2014.
- [52] González A, Aliagas C, Valero M. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. Int. Supercomputing 25th Anniversary*, June 2014, pp.217-226.
- [53] Zhang Y, Soundararajan G, Storer M W et al. Warming up storage-level caches with bonfire. In *Proc. the 11th USENIX Conf. File and Storage Technologies*, Feb. 2013, pp.59-72.



**Xiao-Dong Meng** is a Ph.D. student of computer science in Shanghai Jiao Tong University. He received his Bachelor's degree in electronic information science and technology from Wuhan University in 2007, and Master's degree in information technology from Monash University, Melbourne, in 2010.

His main interest is in parallel and distributed computing, social graph processing, and storage systems.



**Chen-Tao Wu** received his Ph.D. degree in electrical and computer engineering from Virginia Commonwealth University, Virginia, in 2012, and M.E. degree in software engineering in 2006 and B.E. degree in computer science and technology in 2004, both from Huazhong University of Science and

Technology, Wuhan. He is currently an assistant professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, Shanghai. His research interests include computer architecture and data storage systems.



**Min-Yi Guo** received his B.S. and M.E. degrees in computer science from Nanjing University, Nanjing, in 1982 and 1986, respectively, and his Ph.D. degree in information science from University of Tsukuba, Tokyo, 1998. He was a full professor at The University of Aizu and is the head of Department

of Computer Science and Engineering at Shanghai Jiao Tong University. His main interests include automatic parallelization and data-parallel languages, bioinformatics, compiler optimization, high performance computing, and pervasive computing.



**Jie Li** received his B.E. degree in computer science from Zhejiang University, Hangzhou, in 1982, M.E. degree in electronic engineering and communication systems from China Academy of Posts and Telecommunications, Beijing, in 1985, and Dr. Eng. degree from

the University of Electro-Communications, Tokyo, in 1993. Since April 1993, he has been with the Institute of Information Sciences and Electronics, University of Tsukuba, Ibaraki, where he is currently an associate professor. He is also a chair professor in Shanghai Jiao Tong University. His research covers computer networks, distributed, parallel and mobile systems, and modeling and performance evaluation.



**Xiao-Yao Liang** received his Ph.D. degree in electrical engineering from Harvard University in 2008. He is a professor and the associate dean of the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai. His research

interests include computer processor and system architectures, energy efficient and resilient microprocessor design, high throughput and parallel computing, general purpose graphic processing unit (GPGPU) and hardware/software co-design for the cloud and mobile computing. He is also interested in IC design, VLSI methodology, FPGA innovations and compiler technology. He has ample industry experience working as a senior architect or IC designer at companies like NVIDIA, Intel, and IBM.



**Bin Yao** received his B.S. degree and M.S. degree in computer science from the South China University of Technology, Guangzhou, in 2003 and 2007, respectively, and his Ph.D. degree in computer science from the Florida State University, Tallahassee, in 2011.

He has been an associate professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University since 2014. His research interests are management and indexing of large databases, and scalable data analysis.



**Long Zheng** received his B.S. degree in computer science and technology from Huazhong University of Science and Technology (HUST), Wuhan, in 2006, his M.S. degree in computer science and engineering from the University of Aizu, Aizu-Wakamatsu, in 2009,

and his M.S. degree in computer science and technology from HUST, Wuhan, in 2010. He was a visiting scholar with Embedded and Pervasive Computing Center at Shanghai Jiao Tong University, Shanghai, from October 2010 to March 2011. He received his Ph.D. degree from the University of Aizu, 2014. He has been a researcher in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, since 2014. His research interests include chip multiprocessor, parallel and distributed processing, and pervasive computing.