CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3530

SPECIAL ISSUE PAPER

AMC: an adaptive multi-level cache algorithm in hybrid storage systems

Yuxia Cheng[1,*,†], Wenzhi Chen[1], Zonghui Wang[1], Xinjie Yu[1] and Yang Xiang[2]

[1]*College of Computer Science and Technology, Zhejiang University, 866 Yuhangtang Road, Hangzhou, 310058, China*
[2]*School of Information Technology, Deakin University, 221 Burwood Highway, Burwood, VIC, 3125, Australia*

SUMMARY

Hybrid storage systems that consist of flash-based solid state drives (SSDs) and traditional disks are now widely used. In hybrid storage systems, there exists a two-level cache hierarchy that regard dynamic random access memory (DRAM) as the first level cache and SSD as the second level cache for disk storage. However, this two-level cache hierarchy typically uses independent cache replacement policies for each level, which makes cache resource management inefficient and reduces system performance. In this paper, we propose a novel adaptive multi-level cache (AMC) replacement algorithm in hybrid storage systems. The AMC algorithm adaptively adjusts cache blocks between DRAM and SSD cache levels using an integrated solution. AMC uses combined selective promote and demote operations to dynamically determine the level in which the blocks are to be cached. In this manner, the AMC algorithm achieves multi-level cache exclusiveness and makes cache resource management more efficient. By using real-life storage traces, our evaluation shows the proposed algorithm improves hybrid multi-level cache performance and also increases the SSD lifetime compared with traditional multi-level cache replacement algorithms. Copyright © 2015 John Wiley & Sons, Ltd.

Received 3 March 2015; Accepted 15 March 2015

KEY WORDS: hybrid storage; solid state drive; multi-level cache; adaptive algorithm

1. INTRODUCTION

Storage systems have been changing rapidly over the past few years. Storage system architectures are evolving quickly with several fundamental shifts occurring. First, flash-based solid state drives (SSDs) have been widely used not only in the enterprise-class storage systems but also in personal computer systems. Second, disk-based network storage arrays are embracing hybrid, multi-level, and caching-based models. Third, storage systems directly attached to local computing nodes are back in vogue. There are also some two-level hybrid-caching systems [1–4] that exist today and locally attached to computing nodes with dynamic random access memory (DRAM) as the first level cache and SSD as the second level cache for the slower disk hard disk drive (HDD) storage.

The hybrid storage systems (DRAM–SSD–HDD) take advantage of the SSD features such as low cost/gigabyte and non-volatility compared to DRAM and low access latency compared to HDD. The design of such a two-level-caching model is beneficial to the overall performance of the hybrid storage system. Typically, the DRAM and SSD cache levels are managed independently with each level using traditional single level cache management algorithms such as the least recently used (LRU) algorithm.

*Correspondence to: Yuxia Cheng, College of Computer Science and Technology, Zhejiang University, 866 Yuhangtang Road, Hangzhou, 310058, China.
†E-mail: rainytech@zju.edu.cn

However, previous studies [5–7] have shown that the level-independent cache management in multi-level cache systems faces two major drawbacks: (1) Weakened locality in the second level cache. Application access requests, as seen by the second level cache, are filtered by the first level cache. (2) Data redundancy. Because caches are independently managed, data blocks that are cached in the first level have a high probability of being cached in the second level as well. Therefore, data blocks cached redundantly in both levels cause a waste of cache space. This also reduces the total hit ratio.

To address these problems, researchers [5, 8] have proposed multi-level exclusive caching policies. In multi-level exclusive caching systems, data blocks are exclusively cached in one of the cache levels, and caches are managed uniformly or cooperatively as opposed to independently. Recent research [7] has demonstrated that multi-level exclusive caching shows better performance than independent cache management in hybrid storage systems.

Multi-level exclusive caching techniques are commonly used in distributed caching systems [9, 10], where all cache levels use DRAM as caches and are connected by a network. The multi-level exclusive caching policies, when used in the locally attached hybrid caching systems, should consider some specific problems.

(1) Hybrid caches should take care of SSD lifetime where previous distributed caching systems have not taken it into consideration.
(2) Hybrid caches should also consider both read/write operations where most previous researches of distributed multi-level cache ignore caching dirty blocks.
(3) Hybrid caches should be managed in a more integrated way than distributed cache systems. Distributed cache systems (caches are loosely connected via networks) have network bandwidth bottleneck [11]. Locally attached hybrid caches eliminate the network bandwidth bottleneck and provide opportunities for closer collaboration among different cache levels.

In this paper, we propose a novel adaptive multi-level cache (AMC) replacement algorithm in the locally attached hybrid cache systems. The contributions are as follows:

(1) The AMC algorithm introduces combined selective promote and demote operations to dynamically determine the level in which the blocks are to be cached. AMC achieves cache exclusivity using promote and demote operations. By adding the 'selective' property into the promote operation, AMC accumulates more cache hits in DRAM. By adding the 'selective' property into the demote operation, AMC evicts less useful blocks to increase SSD lifetime.
(2) We design an online adaptation method using probabilistic promote and demote values. The probabilistic values control the selective rate of determining the level in which the blocks are to be cached. These values are adjusted according to the usage of blocks already cached in both levels.
(3) We both consider read and write operations in the AMC algorithm. AMC is designed as a unified read–write multi-level cache algorithm to more effectively explore data locality information.

By using real-life storage traces, our evaluation shows the proposed AMC algorithm reduces average response time by up to 25% and increases SSD lifetime by up to 4.12 times compared with traditional multi-level cache replacement algorithms.

The rest of this paper is organized as follows. In Section 2, we describe related work about multi-level caches in hybrid storage systems. Section 3 presents a detailed description of our proposed AMC replacement algorithm. Section 4 reports the experimental results for performance evaluation. Finally, we conclude this paper in Section 5.

2. RELATED WORK

Hybrid storage systems are more commonly used as flash-based SSD technology matures. Researchers [12, 13] have shown great interest in improving the performance and lifetime of SSD, and much research has focused on the optimization problem of using SSD as the disk cache.

DOI: 10.1002/cpe

Kgil *et al.* [14] proposed to improve performance and reliability of flash-based disk caches and also to reduce the power consumption. Pritchett *et al.* [15] proposed an ensemble-level disk cache for cost-performance and introduced a sieving mechanism to reduce SSD allocation writes. Yongseok Oh *et al.* [16] proposed a dynamic garbage collection scheme to improve performance of hybrid storage systems. [4] and [17] proposed novel block management policies to improve performance of a flash-based disk cache. These proposals mainly focus on the single level flash-based disk cache.

There are many other solutions that optimize the flash-based SSD hybrid storage systems. Except for using SSD as disk caches, dynamic storage tiering (DST) technologies [18–21] have been proposed to meet the growing demand for high performance, large capacity, and low cost storage systems. The DST technologies intelligently select frequently accessed data sets and move them into the faster SSD, while the less frequently accessed data sets remain stored in HDD. Other novel techniques [22, 23] have been proposed to make the SSD part of the memory in order to improve performance and reduce cost per gigabyte of memory. [3] and [24] also uses the flash-based SSD as an extended buffer pool for database systems.

In this paper, we focus on the multi-level cache replacement algorithms in the hybrid storage system. Single-level cache replacement algorithms have been extensively studied for decades. Many cache replacement algorithms such as 2Q [25], least recently/frequently used [26], low inter-reference recency set [27], adaptive replacement cache (ARC) [28], and so on were proposed to achieve a better hit ratio than the traditional LRU replacement algorithm. These algorithms were designed and used in the single level cache scenario but did not take multi-level cache hierarchies into account.

Previous researchers [5, 8–11] have studied the multi-level cache management techniques in the context of distributed multi-tier storage systems, where the first level cache typically resides in the front end application servers and the second level cache resides in the networked storage servers. Wong *et al.* [5] proposed a method to eliminate data redundancy by applying a unified LRU scheme to achieve exclusive caching. They introduced a demote operation to transfer data ejected from the client cache to the storage cache. However, the demote operation incurs high network traffic and system overhead to prepare, send, and receive demoted blocks. Eviction-based cache placement [11] policy was then proposed to decrease network bandwidth usage. This policy uses a client content tracking table to estimate a client's block eviction information, and this information is then sent to the storage cache, where the storage cache then reloads the evicted blocks from disks into its cache. This eviction-based policy incurs extra I/Os on disks and increases the average miss penalty. In distributed storage systems, the network bandwidth problem is a major performance bottleneck. Chen *et al.* [6] systematically studied a large design space of distributed multi-level cache management.

However, the design of the hybrid multi-level cache algorithm has to consider some specific aspects that are different from the distributed multi-level cache algorithms we described in Section 1. Recent studies have tried to introduce the multi-level exclusive cache into hybrid storage systems. Raja *et al.* [7] have investigated the design tradeoffs involved in building exclusive, direct-attached, multi-level storage caches. They demonstrated the potential performance benefits of maintaining multi-level cache exclusivity in the multi-level hybrid cache system. In this paper, we further proposed a novel AMC algorithm that uses combined selective promote and demote operations to dynamically determine the level in which the blocks are to be cached. The algorithm considers both multi-level cache exclusivity and SSD lifetime.

3. ADAPTIVE MULTI-LEVEL HYBRID CACHING

In this section, we will describe the design of our adaptive multi-level hybrid cache algorithm and explain how the algorithm achieves cache exclusiveness using the selective promote and demote operations. Then, we describe how the algorithm adapts itself according to its online cache status. We also describe how the algorithm handles dirty blocks.

DOI: 10.1002/cpe

3.1. Overview

The key idea of AMC is to determine the appropriate level in which new blocks will be cached and the level from which old blocks will be evicted. To achieve this, AMC uses selective promote and demote operations. The promote and demote operations are used in multi-level cache scenarios to collaboratively maintain data exclusivity among different cache levels. Using a selective promote operation, AMC can choose to put new blocks whether in either DRAM or SSD caches. Using a selective demote operation, AMC can choose to evict old blocks from either the DRAM or SSD caches. These selective operations depend on the current status of two caches. AMC uses the average lifetime of blocks in the LRU end of a cache to represent the current status of the cache (named marginal life utility). AMC further uses probabilistic values to reflect the difference of marginal life utility values between caches, and these values are adjusted online according to cache status. The probabilistic values indicate the tendency of the selective operations-caching blocks in which level. The selective promote and demote operations are based on the probabilistic values. For example, if the marginal life utility of a SSD cache becomes larger than that of a DRAM cache, more new blocks will tend to be put into the SSD cache to evict old blocks in the same level. If the marginal life utility of the DRAM cache becomes larger, more new blocks will tend to be put into the DRAM cache, and the old blocks in DRAM will be evicted from the cache. In this way, new blocks are cached into appropriate levels, and old blocks are evicted from the corresponding caches.

3.2. Achieving exclusivity

Figure 1 shows a two-level DRAM–SSD hybrid cache hierarchy. The first level cache is DRAM level 1 cache (L1 cache); the second level cache is SSD level 2 cache (L2 cache). The L1 list and L2 list in the figure maintains the metadata of blocks (metadata contains a block's identifier and other attributes) cached in the L1 and L2 caches, respectively. $L1_{temp}$ list and $L2_{ghost}$ list maintain the metadata of some accessed blocks used for the adaptive algorithm, which will be described in

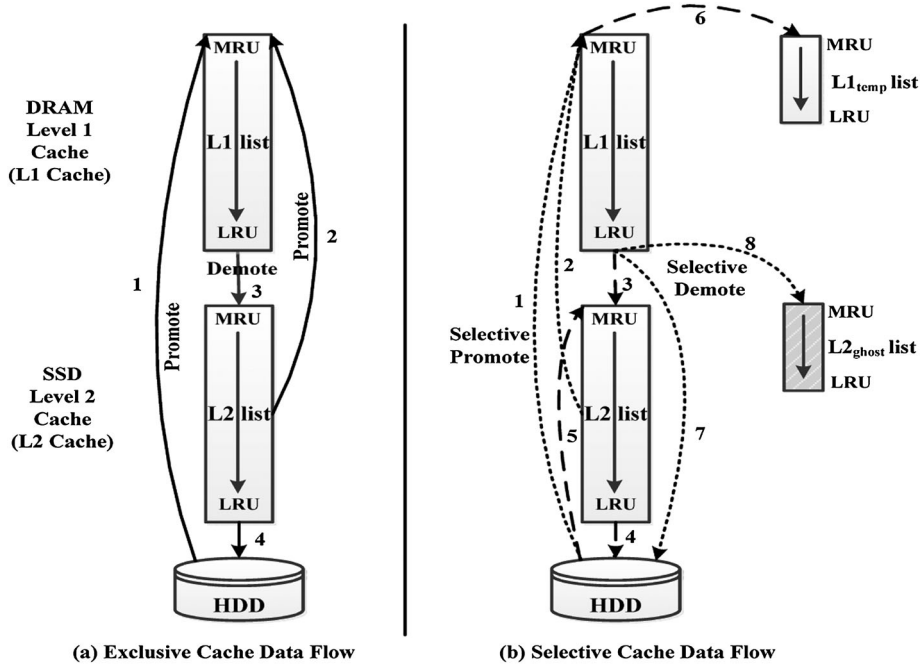


Figure 1. The two-level hybrid cache hierarchy. (a) Traditional two-level exclusive cache. Arrow lines show block data flows. Lines 1 and 2 represent block promote operations; lines 3 and 4 represent block demote operations. (b) Selective promote/demote-based two-level cache structure. Lines 1, 2, 5, and 6 represent the selective promote operations. Lines 3, 4, 7, and 8 represent the selective demote operations. DRAM, dynamic random access memory; MRU, most recently used; SSD, solid state drive; LRU, least recently used; HDD, hard disk drive.

DOI: 10.1002/cpe

a later subsection. We first introduce two operations (demote and promote operations as shown in Figure 1(a)) used in multi-level cache hierarchies to achieve exclusivity.

The demote operation is used when the L1 cache is full and needs to replace the LRU block in the L1 list for the new most recently used (MRU) block. The LRU block in the L1 list denotes the block's metadata is in the LRU position in the L1 list, and the block's data are cached in the L1 cache. In the demote operation, the LRU block in the L1 list is evicted from the L1 cache and inserted the block's metadata into the MRU position in the L2 list and also allocated a free cache space for the block. If the L2 cache is full, the LRU block in the L2 list must be evicted to make room for the demoted block. We call the blocks evicted from the L1 cache and demoted into the L2 cache the demoted blocks.

The promote operation is used when a block miss occurs in the L1 cache. The missed block may be cached in the L2 cache or may reside in the HDD. In the promote operation, the missed block is fetched from the L2 cache (also invalidate the block in L2 cache and delete the block's metadata from L2 list) or the HDD and inserted into the MRU position in the L1 list and also allocated a free cache space in the L1 cache. If the L1 cache is full, a demote operation is needed to make room for the missed block. We call blocks promoted into the L1 cache the promoted blocks.

In order to achieve exclusivity in multi-level caches, one traditional exclusive cache algorithm is as follows. Every time a new block request arrives, we make sure the block is only allocated in the L1 cache until the L1 cache is full (promote operation). Allocation in the second level cache only occurs during eviction from the first level (demote operation). In addition, any first level cache misses that find the data cached in the second level result in the data being deleted from the second level and allocated in the first level (promote operation). This is a straight forward method to achieve cache exclusivity.

However, we find two major problems in the traditional exclusive caching algorithm that motivate the design of selective promote and demote operations. First, if each new block miss in the L1 cache results in cache allocation in the L1 cache, this means all block misses incur promote operations, which may cause some older but 'hot' blocks cached in L1 to be evicted (promote operations need subsequent demote operations when the L1 cache is full). Second, if each block evicted from the L1 cache is demoted into the L2 cache, this will cause a large number of write allocations in the SSD cache and may also evict 'hot' blocks originally cached in L2. To address these problems, we introduce the selective promote and demote operations to more effectively keep 'hot' blocks in both the DRAM and SSD caches while still achieving cache exclusivity.

3.3. Selective promote/demote

Figure 1(b) shows the selective cache data flow. In general, the selective promote operation controls the rate of new blocks coming into DRAM and SSD caches to prevent hot blocks being evicted. The selective demote operation determines from which level the cold blocks should be evicted. The detailed selective promote and demote operations are described as follows.

When a new block request arrives, the algorithm determines which level the blocks should be cached and replaces the less useful block in the multi-level cache hierarchy. The main algorithm is shown in Figure 2. If a block hit occurs in the L1 cache, then the LRU replacement algorithm is used as normal (case 1, lines 1–2). If case 1 does not occur (cases 2–5), then before allocating a new free space for the missed block in the L1 cache, the algorithm first checks the current DRAM and SSD cache status based on their marginal life utility represented by the probabilistic value (selective_promote(X), lines 5,7,10, and 12), which determines the level the new block should be placed into. As Figure 3 shows, if the block passes the check, the algorithm allocates the block in the L1 cache (lines 14–15). Otherwise, the algorithm puts the block in the L2 cache (line 18, the function of line 17 will be described in Section 3.5).

Similarly, before allocating a new free space in the L2 cache to make room for the block evicted from the L1 cache (the demote operation), the algorithm first checks whether this block is suitable to be cached in L2 (selective_demote(), line 22). If the block passes the check, the algorithm then puts the block in the L2 cache (lines 26–27). Otherwise, the block is just deleted from the L1 cache and never cached in L2 (lines 28–29, $L2_{ghost}$ is a ghost cache described in Section 3.5. We first focus on

DOI: 10.1002/cpe

```
/* Upon a reference to the block X, */
/* Only one of the following five cases must occur */
1:  Case 1: X is in L1                          //cache hit in DRAM
2:          do_LRU(L1, X);
3:  Case 2: X is in L1temp                      //cache hit in DRAM
4:          update_probability( );
5:          selective_promote(X);
6:  Case 3: X is not in DRAM but in L2          //cache hit in SSD
7:          selective_promote(X);
8:  Case 4: X is in L2ghost                     //cache miss
9:          update_probability( );
10:         selective_promote(X);
11: Case 5: X is not in the above 4 cases       //cache miss
12:         selective_promote(X);
```

Figure 2. Adaptive multi-level hybrid cache management algorithm. DRAM, dynamic random access memory; SSD, solid state drive; LRU, least recently used; L1, first level; L2, second level.

```
/* Subroutines, selective Promote & Demote operations */
13: selective_promote(X)
14:     if(promote_check())
15:         promote_adjust(X);
16:     else
17:         do_LRU(L1temp, X);
18:         do_LRU(L2, X);
19: promote_adjust(X)
20:     Delete X in L2 or L1temp or L2ghost(if exist)
21:     if(L1 is full)
22:         selective_demote();
23:     do_LRU(L1, X);
24: selective_demote( )
25:     Delete LRU block X' in L1 list
26:     if(demote_check())
27:         do_LRU(L2, X');
28:     else
29:         do_LRU(L2ghost, X');
```

Figure 3. Subroutines of the selective promote and demote operations. LRU, least recently used; L1, first level; L2, second level.

clean blocks to ease the algorithm description, and the dirty block's management will be described in Section 3.6). The rationale behind this is that blocks evicted from the L1 cache are not always more useful than blocks already cached in the L2 cache [8, 15]. Thus, simply deleting these blocks will reduce the number of SSD allocations and will prevent replacing out blocks cached in L2.

3.4. Adaptive probability

In this section, we describe an online adaptive method of tracking the status of caches using dynamically adjusted probabilistic values, which help the selective promote and demote operations determine the cache data flow.

DOI: 10.1002/cpe

```
        /* Subroutines, Adaptive Probabilities*/
30:  promote_check( )
31:      return rand( ) < P_promote ?   True : False;
32:  demote_check( )
33:      return rand( ) < P_demote ?   True : False;
34:  update_probability( )
35:      MU_1 = update_life_utility(L1);
36:      MU_2 = update_life_utility(L2);
37:      P_promote += (1- P_promote) * P_promote* ( MU_1/(MU_1+MU_2) - 1/2 ) ;
38:      P_demote += (1- P_demote) * P_demote* ( MU_2/(MU_1+MU_2) - 1/2 ) ;
39:  update_life_utility( L )
40:      return (sum_{i=1}^{n} lifetime_i)/n   //calculate utility on the last n
41:                                            //blocks in the LRU ends of L list
```

Figure 4. Subroutines of the adaptive probabilities adjustment. LRU, least recently used.

We use the average lifetime of blocks in the LRU end of a cache to represent the current status of the cache (named marginal life utility). A block's lifetime is the current time minus the time the block last accessed. Thus, the marginal life utility is calculated by averaging the lifetime of the last n blocks in the LRU end of the cache list (Figure 4, lines 40–41); n is a configurable parameter that is usually set to a small fixed number [29] (we set $n = 10$; any other small values would work as well). The larger marginal life utility value of a cache, the less useful blocks reside in the LRU end of the cache. To reflect the difference of marginal life utility between caches, we introduce two probabilistic values $P_{promote}$ and P_{demote} ($0 < P_{promote}, P_{demote} < 1$).

The values of $P_{promote}$ and P_{demote} indicate the relative degree of the old blocks' usefulness between two cache levels and help determine new blocks should be cached in which level (lines 30–33). As Figure 4 shows, the algorithm compares marginal life utilities between two cache levels and periodically adjusts the $P_{promote}$ and P_{demote} values (lines 35–38) online. If the marginal life utility of the L1 cache (MU_1) is larger than the marginal life utility of the L2 cache (MU_2), that is $(\frac{MU_1}{MU_1+MU_2} - \frac{1}{2}) > 0$ or $(\frac{MU_2}{MU_1+MU_2} - \frac{1}{2}) < 0$, then we increase the $P_{promote}$ value and decrease the P_{demote} value (lines 37–38). Similarly, if MU_1 is smaller than MU_2, then we decrease the $P_{promote}$ value and increase the P_{demote} value. The goal of selective operations is to equalize the marginal life utility of two caches. If the values of $P_{promote}$ and P_{demote} approach $\frac{1}{2}$, the marginal life utility of two caches is roughly the same.

Upon referencing a new block, the probability of the block being promoted into the L1 cache depends on the $P_{promote}$ value (line 30–31, the rand() function returns a random decimal number between 0 and 1). We can see the smaller $P_{promote}$ value, the lower the rate at which new blocks will pass the promote check (when promote_check() returns true) and enter the L1 cache. The same is true for the P_{demote} value. The smaller the P_{demote} value, the lower the rate at which the evicted blocks from L1 will enter the L2 cache. For example, if $P_{promote}$ approximates 1, then almost all new blocks will be promoted into the L1 cache. If $P_{promote}$ approximates 0, then few blocks will be promoted and thus be placed into the L2 cache. Therefore, we can conveniently adjust $P_{promote}$ and P_{demote} values to control the rate of new blocks coming into the L1 and L2 caches, respectively.

The rationale behind this is that when MU_1 is smaller than MU_2, it means the L1 cache has more useful blocks in the LRU end of the L1 list than the L2 cache. Therefore, decreasing the $P_{promote}$ value will decrease the probability of new blocks flushing into the L1 cache and will increase the probability of new blocks flushing into the L2 cache, thus preventing more useful blocks in the LRU end of the L1 list being replaced by new blocks. At the same time, the P_{demote} value is increased in

DOI: 10.1002/cpe

case more useful blocks in L1 are evicted (relative to the LRU blocks in the end of L2 list), the larger the probability they will be demoted into the L2 cache and vice versa. This means the algorithm keeps more useful blocks in the caches and evicts less useful blocks. On each block replacement, the algorithm tries to evict the LRU block in the cache that has larger marginal life utility.

The $P_{promote}$ and P_{demote} values are adjusted proportionally to the difference between MU_1 and MU_2, and the adaptation becomes slower when the $P_{promote}$ and P_{demote} values are close to extreme values of 0 and 1 (the product factor form of (1-P)*P). The $P_{promote}$ value is initialized to the ratio of the L1 cache size divided by the aggregate cache size (L1 cache size plus L2 cache size), and the P_{demote} value is initialized to the ratio of the L2 cache size divided by the aggregate cache size. The algorithm updates the $P_{promote}$ and P_{demote} values periodically to let the selective promote and demote operations take effect. For example, every 100 block accesses the algorithm invokes update_probability(). The algorithm also updates the two values when a block reference hits $L1_{temp}$ and $L2_{ghost}$, which will be described in the following section.

3.5. Temp cache and ghost cache

The MRU block may not be cached in L1 because of the block failing the check (lines 16–18) in the selective promote operation. In case the block is re-accessed in the near future, we introduce a small temporary cache ($L1_{temp}$, named temp cache) to temporarily keep those un-promoted blocks in the L1 cache (line 17). If these blocks in $L1_{temp}$ are referenced again in the near future, they will have a larger probability of passing the promote check (lines 3, 5, and 14). The principle of the check operation is based on the probabilistic comparison. The more hits incurred by a block, the more it can pass the promote check and finally be cached in the L1 cache. When blocks in $L1_{temp}$ are accessed, the algorithm will adaptively increase the possibility of letting new blocks be promoted into the L1 cache (lines 3–4). The temp cache is used for reducing the cost when the selective promote operation does not promote the block into the L1 cache, which will be quickly re-accessed again. If the un-promoted block in $L1_{temp}$ is not accessed for a period of time, it will be quickly evicted from $L1_{temp}$ and deleted from the cache. We set the size of $L1_{temp}$ relatively small (0.1% of the L1 cache size). The experimental results show the overall cache performance is insensitive to the small size of $L1_{temp}$.

In the selective demote operations, the blocks may simply be deleted from the L1 cache and may fail the check to be demoted into the L2 cache (lines 25–29). If the subsequent I/O requests go to those un-demoted blocks, both L1 and L2 will have cache misses. In order to more accurately determine which blocks should be demoted into the L2 cache, we introduce a ghost cache to keep un-demoted blocks in the $L2_{ghost}$ list (line 29). The ghost cache only maintains blocks' metadata, and many other cache algorithms [25, 27, 28] use the ghost cache to capture additional data access patterns. We set the size of the $L2_{ghost}$ relatively small (0.1% of the L2 cache size, as similar to the size of $L1_{temp}$). If blocks in the $L2_{ghost}$ are frequently accessed in a certain period (lines 8–9), the algorithm will adaptively increase the possibility of letting the LRU block in L1 list pass the demote check to insert the block into the L2 cache, thus increasing the cache hits of demoted blocks in SSD cache. The adaptive adjustment process is described in the previous section.

3.6. Handling writes

In this section, we discuss how to handle dirty blocks and integrate them into the proposed multi-level hybrid cache algorithm. To manage dirty blocks and block write operations in multi-level cache hierarchies, we have to address two major issues. One is data consistency, and the other is performance overhead.

In order to achieve data consistency among multi-level cache hierarchies, we make sure every dirty block is only cached in one level. The proposed AMC algorithm achieves exclusivity between two cache levels except for a small temporary cache ($L1_{temp}$) in DRAM (non-strict exclusivity for clean blocks). When handling dirty blocks, the algorithm maintains strict exclusivity using write allocation and fetch-on-write policies. When writing a new block, the algorithm directly allocates

DOI: 10.1002/cpe

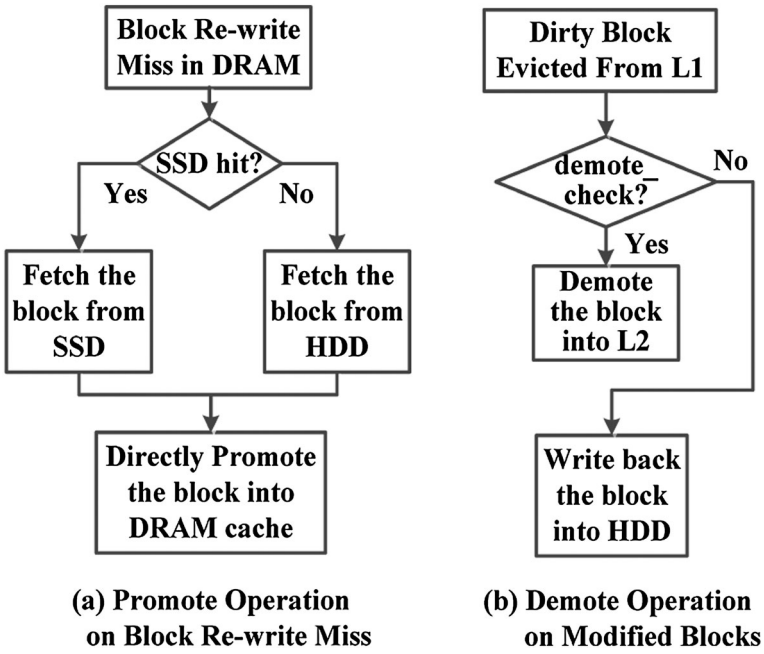


Figure 5. Handling writes under the selective promote–demote operations. DRAM, dynamic random access memory; SSD, solid state drive; HDD, hard disk drive; L1, first level; L2, second level.

the block in the DRAM cache. Figure 5(a) shows how to handle a block re-write (write request on an existing block) miss under the selective promote operation. If a block re-write misses in the L1 cache (cases 2, 3, 4, and 5 in the previous section), we skip the promote_check() operation and directly promote the block (which could be in the SSD cache or HDD) into the L1 list to make sure the written block is only cached in one level. In the selective demote operation, as Figure 5(b) shows, if the block evicted from the L1 cache has been modified but failed the demote_check() operation, we write back the modified block into the HDD directly. Otherwise, the block is demoted into the L2 cache. Dirty blocks in both DRAM and SSD caches should be flushed back into the HDD periodically. Through maintaining exclusivity of dirty blocks among cache hierarchies, we achieve data consistency by design.

The algorithm is designed as a unified read–write multi-level cache, with the data locality information more effectively maintained than the separate read and write cache regions. When handling dirty blocks, we take advantage of the selective operations. In the selective promote operation, we promote all written blocks into the L1 cache because written blocks usually exhibit higher locality [17]. In the selective demote operation, the modified blocks evicted from L1 are intelligently filtered by their marginal life utility. The more useful blocks are demoted into the SSD cache, while the less useful blocks are written back into the HDD directly. In a write dominant scenario, the selective promote operation is seldom triggered. However, the selective demote operation can still bring the benefit of reducing the number of SSD writes.

4. PERFORMANCE EVALUATION

In this section, we use the trace-driven simulation method to evaluate the proposed adaptive multi-level hybrid cache algorithm. We implement multi-level hybrid cache hierarchies in a simulator originated from [30]. We use the simulator to benchmark cache algorithms using a wide variety of real-life enterprise storage traces [31].

In order to analyze multi-level hybrid cache management algorithms, we implemented a trace-driven simulator that consists of the first level DRAM cache, the second level SSD cache, and the disk. The simulator has three major parts: the parser, the hash manager, and the cache algorithm module. The parser is used to convert traces of a different format into the simulator's customized

DOI: 10.1002/cpe

Table I. Different types of storage traces.

Trace Name	Blocks ($\times 10^3$)	Requests ($\times 10^3$)	Read (%)	Write (%)
WebSearch1	2147	3996	99.99	0.01
WebSearch2	3767	17,255	99.99	0.01
WebSearch3	3680	32,832	99.95	0.05
Financial1	818	6968	19.23	80.77
Financial2	471	4480	79.52	20.48
Zipf	1000	6000	80.00	20.00

format and simulating I/O requests based on the input trace file. The hash manager is a set of common routines used by different cache algorithms to maintain cached blocks' information for quick search, insert, and delete. The cache algorithm module implements various multi-level cache algorithms. The simulator is initialized using cache sizes, trace format, and algorithm type as input.

The simulator passes each I/O request (generated by the parser) into the cache algorithm module and records various statistics during the simulation for algorithm analysis. In our trace-driven simulation environment, we simulate realistic response time for multi-level hybrid cache hierarchies. We calculate the average response time (*avg_time*) in the multi-level hybrid cache simulation using the following equations:

$$\begin{aligned} virtual\_time = {} & SSD_{reads} * t_{SSDr} + SSD_{writes} * t_{SSDw} \\ & + HDD_{reads} * t_{HDDr} + HDD_{writes} * t_{HDDw} \end{aligned} \tag{1}$$

$$avg\_time = \frac{virtual\_time}{|I/O\ requests|} \tag{2}$$

We first calculate the total virtual execution time (*virtual_time*) as equation (1) demonstrates then use the *virtual_time* to calculate the *avg_time* as equation (2) shows. SSD_{reads} (SSD_{writes}) means the total number of reads (writes) in the SSD cache, and t_{SSDr} (t_{SSDw}) is the average read (write) latency of SSD. Similarly, HDD_{reads} (HDD_{writes}) means the total number of reads (writes) in the HDD, and t_{HDDr} (t_{HDDw}) is the average read (write) latency of HDD. The average response time of each I/O requests in a certain trace is then calculated using *virtual_time* divided by the total number of I/O requests. Because the total time taken by DRAM hits is negligible compared with the relative large latency of SSD and HDD, we do not calculate it in our response time model. We set $t_{SSDr} = 25\mu s$, $t_{SSDw} = 200\mu s$, and $t_{HDDr} = t_{HDDw} = 5ms$. These settings model the behavior of traditional HDD and SSD, which are identical to those used in prior research [7, 16].

We use real-life enterprise storage traces that have been widely used for evaluating caching algorithms as shown in Table I. The WebSearch1, WebSearch2, and WebSearch3 are traces collected by monitoring I/O requests of search engine applications. All of the aforementioned three traces are read dominant traces. The Financial1 and Financial2 are traces collected by monitoring I/O requests of online transaction processing applications running at two large financial institutions. Both the two traces are read–write traces, and the Financial1 is the write dominant trace. Zipf is a synthetic trace (20% writes) that follows Zipf-like distribution where the probability of the i^{th} block being accessed is proportional to $1/i^{\alpha}$ ($\alpha = 0.75$). This approximates some applications where a few blocks are frequently accessed while others are accessed much less frequently.

4.1. Average response time

To demonstrate the performance of the proposed AMC algorithm, we use the average response time (*avg_time*) described in the previous section as a comprehensive evaluation metric. The *avg_time* includes not only the time taken by cache hits and misses but also includes the time taken by writing the demoted blocks into the SSD cache (included in SSD_{writes}). In the experiment, we set the same size for DRAM and SSD caches. We will analyze the impact of varying the DRAM–SSD size ratio in Section 4.4.

DOI: 10.1002/cpe

There are four comparative multi-level cache algorithms in the figure. One is the AMC-LRU algorithm that we have proposed in this paper. We call it AMC-LRU because the algorithm is based on the LRU algorithm. The rest of three algorithms are described as follows:

(1) **ind-LRU** is a two-level independent LRU algorithm. The ind-LRU deploys LRU algorithms independently between the DRAM and SSD caches. Both levels cache blocks on each block miss. When a block request to a DRAM cache results in a miss, the request is passed to the SSD cache. If the block request hits in the SSD, this block is then inserted into the DRAM cache (SSD also maintains the same block), and the LRU block is replaced when the DRAM cache is full. If the requested block is not found in the SSD cache, the block is allocated in both caches.
(2) **exc-LRU** is a two-level exclusive LRU algorithm. When a new block request arrives, the exc-LRU algorithm makes sure the block is only allocated in the DRAM cache until the cache is full. Allocation in the SSD cache only happens during eviction from the DRAM. In addition, any DRAM misses that find the block in the SSD result in the block being deleted from the SSD and allocated in the DRAM.
(3) **2C-LRU** is a one-level LRU algorithm whose cache size equals the aggregate size of two level caches. 2C-LRU only uses the DRAM cache.

Figure 6 shows the average response time (*avg_time*) of four comparative multi-level cache algorithms under six different traces (three read-dominant traces and three read–write traces). We chose the cache size for each trace according to their relative working set size (as shown in Table I). A too large or too small cache sizes will diminish the effect of different cache replacement algorithms. The results shown here are the average results of three runs; the experimental results are stable, and the variation among separate runs is within 0.5%, even for the adaptive AMC algorithm. From the experimental results shown in Figure 6, we can make the following observations.

First, under all six traces, the ind-LRU algorithm has a much larger *avg_time* than the other three algorithms. This is because the ind-LRU manages cache blocks independently that results in a data redundancy problem between two cache levels. Therefore, the total effective cache space

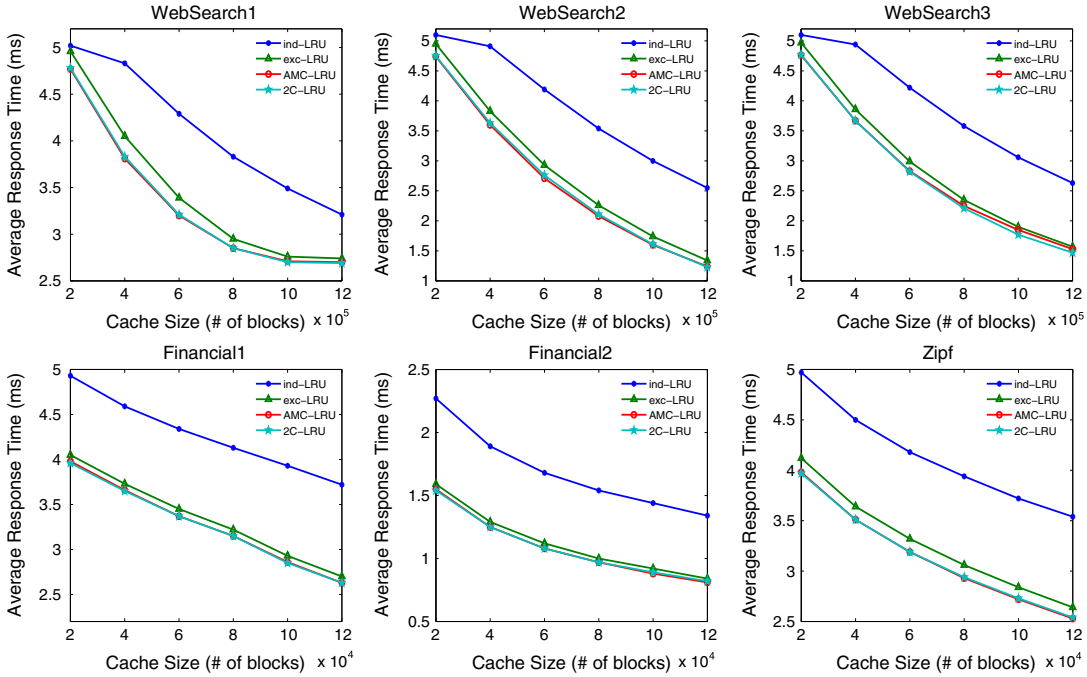


Figure 6. The average response time of four comparative multi-level cache algorithms under three read-dominant traces and three read–write traces. LRU, least recently used; AMC, adaptive multi-level cache.

DOI: 10.1002/cpe

is significantly reduced compared to the exclusive multi-level cache algorithms. For example, the *avg_time* of ind-LRU is 19% higher than exc-LRU and 25% higher than AMC-LRU and 2C-LRU in the WebSearch1 trace.

Second, our proposed AMC-LRU algorithm outperforms ind-LRU and exc-LRU. The AMC-LRU achieves exclusivity between cache levels so it has a larger effective cache space than ind-LRU. Therefore, AMC-LRU has better performance than ind-LRU. The exc-LRU also archives exclusivity, but the AMC-LRU still outperforms exc-LRU. This is because exc-LRU needs a larger number of demote operations than AMC-LRU, which leads to a higher latency of SSD allocations. Increasing the number of SSD allocations also reduces SSD lifetime, and we will present a comparison of SSD allocations in the next section. What is more, the AMC-LRU keeps more useful blocks in both caches than exc-LRU thus has a higher hit ratio than exc-LRU. For example, the *avg_time* of AMC-LRU is 12% shorter than exc-LRU in the WebSearch2 trace.

Third, the performance of the AMC-LRU algorithm approaches the 2C-LRU algorithm. This further demonstrates that AMC-LRU can effectively utilize two-level cache spaces. In the WebSearch2 trace, the AMC-LRU even outperforms 2C-LRU because the total hit ratio of AMC-LRU is higher than 2C-LRU, and the AMC-LRU accumulates large portions of hits in the DRAM cache (Section 4.3). The AMC-LRU algorithm has very few demote operations overhead, which contributes to the performance as good as 2C-LRU. We will evaluate the multi-level hybrid cache algorithms in detail to show the effectiveness of the AMC-LRU algorithm in the following sections.

4.2. *Solid state drive lifetime*

One important design consideration of hybrid cache system is the limited SSD lifetime. The SSD has endurance problems caused by limited erasure counts. Therefore, by reducing the number of SSD allocation times, the lifetime of SSD can be increased. Figure 7 shows the total number of SSD allocations using different multi-level hybrid cache algorithms under six I/O traces. The experimental settings are the same as the previous section. We compare the total number of SSD allocations (SSD writes) of three algorithms: AMC-LRU, exc-LRU, and ind-LRU. In AMC-LRU, the total number of SSD allocations includes the number of demoted blocks into SSD and the number of missed blocks

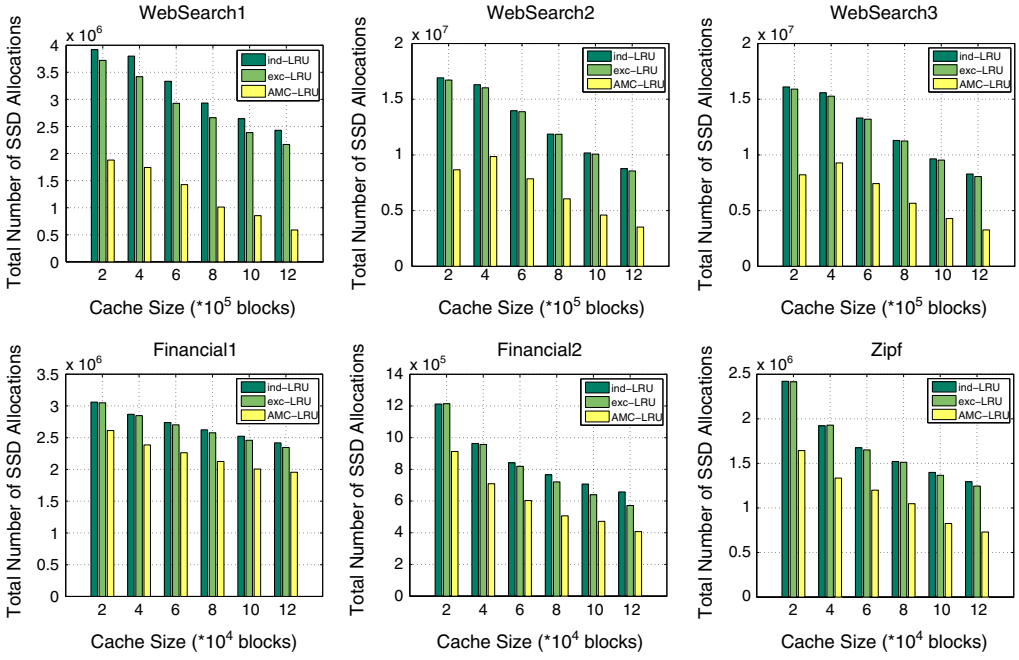


Figure 7. The number of solid state drive (SSD) allocations of four comparative multi-level cache algorithms under three read-dominant traces and three read–write traces. LRU, least recently used; AMC, adaptive multi-level cache.

DOI: 10.1002/cpe

that has not been promoted into DRAM (the un-promoted blocks). In exc-LRU, the total number of SSD allocations is just the number of demoted blocks into the SSD. In ind-LRU, the total number of SSD allocations equals the number of missed blocks in SSD. We do not show 2C-LRU because 2C-LRU has no SSD allocations.

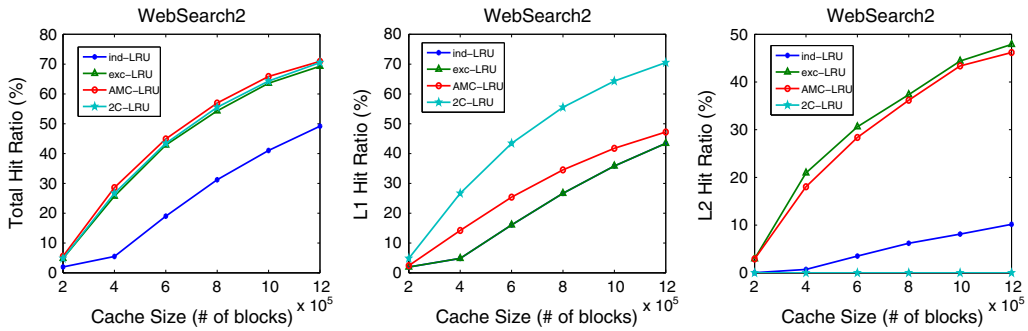
From Figure 7, we find the AMC-LRU algorithm reduces the total number of SSD allocations compared with exc-LRU and ind-LRU. In the read-dominant traces, the number of SSD allocations of ind-LRU and exc-LRU are 1.63 to 4.12 times larger than AMC-LRU. Because AMC-LRU uses the selective demote operation, a large number of blocks evicted from the L1 cache are filtered out by the selective operations and are no longer cached into the SSD. On the other hand, exc-LRU demotes all blocks evicted from the L1 cache into the SSD.

In the read–write traces, the number of SSD allocations of ind-LRU and exc-LRU are 1.17 to 1.77 times larger than AMC-LRU. Due to dirty blocks evicted from the L1 cache could not be simply deleted, the opportunity to reduce the number of SSD allocations becomes smaller in AMC-LRU. Therefore, in read–write traces, the number of SSD allocations reduced by AMC-LRU (compared with ind-LRU and exc-LRU) is smaller than those cases in read-dominant traces. Reducing the total number of SSD allocations can also reduce I/O bandwidth contention and reduce the probability of triggering a garbage collection process inside SSD [16], both of which are critical performance impact factors. Therefore, reducing the number of SSD allocations not only improves SSD lifetime but also contributes to overall improvement in cache performance.

4.3. Hit ratio

In this section, we will analyze the experimental results of different hit ratios among four comparative algorithms. Figure 8(a) shows the total hit ratio curve under the WebSearch2 trace. The total hit ratio is the total hit counts of both the DRAM and SSD caches divided by the number of total block requests. The ind-LRU algorithm has the lowest total hit ratio. The total hit ratios of the other three algorithms, exc-LRU, AMC-LRU, and 2C-LRU, are very close. There is a similar trend in other application traces, however, due to the limited space, we did not show their hit ratio curves. In the WebSearch2 trace, the AMC-LRU has a higher total hit ratio than 2C-LRU. This is the main reason the *avg_time* of AMC-LRU is shorter than 2C-LRU under WebSearch2. A higher total hit ratio results in less DRAM and SSD cache misses, and the cache miss will cause significant latency of a HDD read (5 ms vs. 25 μs).

Figure 8(b) shows the DRAM cache hit ratio curve. We observe the AMC-LRU has a much higher DRAM hit ratio than exc-LRU. This is due to AMC-LRU taking advantage of the selective promote operation and accumulating more useful blocks in the DRAM cache; thus, AMC-LRU achieves a higher DRAM hit ratio than exc-LRU. The ind-LRU and exc-LRU achieve almost the same DRAM



(a) Total Hit Ratio Curve (b) DRAM Cache Hit Ratio Curve (c) SSD Cache Hit Ratio Curve

Figure 8. Comparisons of the total hit ratio, dynamic random access memory (DRAM) cache hit ratio, and solid state drive (SSD) cache hit ratio under the Websearch2 trace: (a) total hit ratio curve, (b) DRAM cache hit ratio curve, and SSD cache hit ratio curve. LRU, least recently used; AMC, adaptive multi-level cache; L1, first level; L2, second level.

DOI: 10.1002/cpe

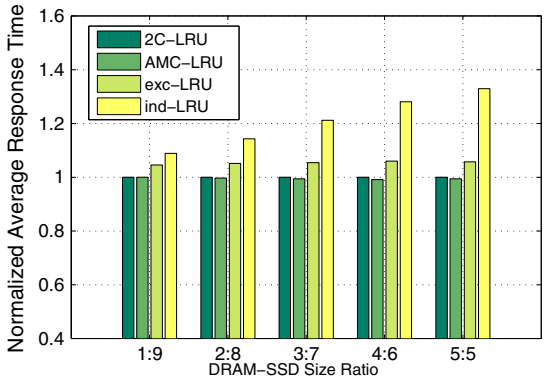


Figure 9. Comparison of average response time with different dynamic random access memory–solid state drive (DRAM–SSD) cache size ratio configurations under the Websearch1 trace. LRU, least recently used; AMC, adaptive multi-level cache.

hit ratio because they both use the same LRU replacement logic in the DRAM cache. The 2C-LRU has the highest DRAM hit ratio because its DRAM cache size is twice as large as the other three algorithms.

Figure 8(c) shows the SSD cache hit ratio curve. We observe the SSD hit ratios of exc-LRU and AMC-LRU are much higher than ind-LRU, which demonstrates the benefit of achieving exclusivity in the hybrid cache scenario. However, the SSD hit ratio of AMC-LRU is lower than exc-LRU. The reason behind this is the selective promote operation of AMC-LRU accumulates more useful blocks in the DRAM cache and lets the un-promoted blocks be cached in SSD. This may cause some one-time accessed blocks to be cached in SSD, which lowers the hit ratio of the SSD cache. This is a major drawback of the LRU algorithm. The SSD hit ratio of 2C-LRU is zero because 2C-LRU only uses the DRAM cache and does not use the SSD cache.

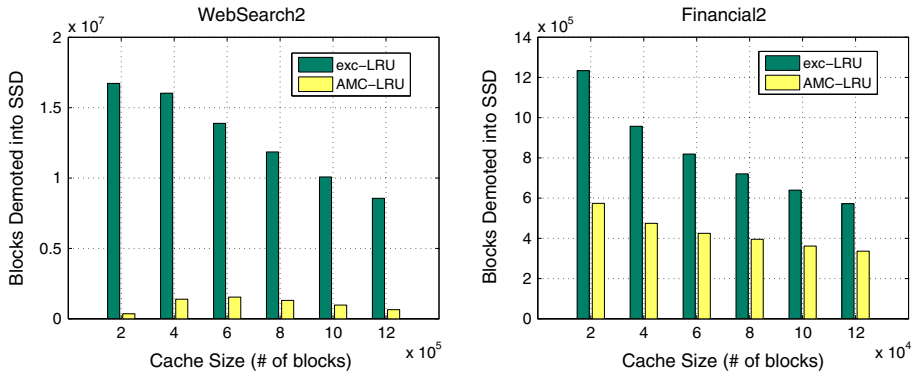
4.4. Dynamic random access memory–solid state drive size ratio

Figure 9 shows the normalized average response time of different DRAM–SSD cache size ratios under the WebSearch1 trace. In order to compare the impact of different DRAM–SSD size ratios on cache performance, we set the aggregate cache size of DRAM and SSD to 20% of the working set size (total number of unique blocks accessed across the entire trace as shown in Table I) and vary the DRAM–SSD cache size ratio from 1:9 to 5:5. The average response time of different algorithms are normalized to the *avg_time* under the 2C-LRU algorithm. The 2C-LRU only uses the DRAM cache, and its cache size is fixed so the *avg_time* of 2C-LRU is the same under different size ratios. The experimental results show the AMC-LRU and exc-LRU algorithms have relatively stable *avg_time* with different DRAM–SSD size ratios. This is because both AMC-LRU and exc-LRU achieve cache exclusivity, and their total hit ratios remain stable as the aggregate cache size is fixed (the *avg_time* is mainly determined by cache misses that incur high latency of HDD reads and the number of demoted blocks that incurs SSD writes). The *avg_time* of AMC-LRU is close to 2C-LRU and is lower than exc-LRU, which is the same as Section 4.1 shows. However, the *avg_time* of ind-LRU increases as the DRAM–SSD size ratio increases from 1:9 to 5:5. In the ind-LRU algorithm, the total effective cache size decreases as the DRAM–SSD size ratio increases up to 5:5. The ind-LRU has a data redundancy problem, as the effective cache size is determined by the larger cache.

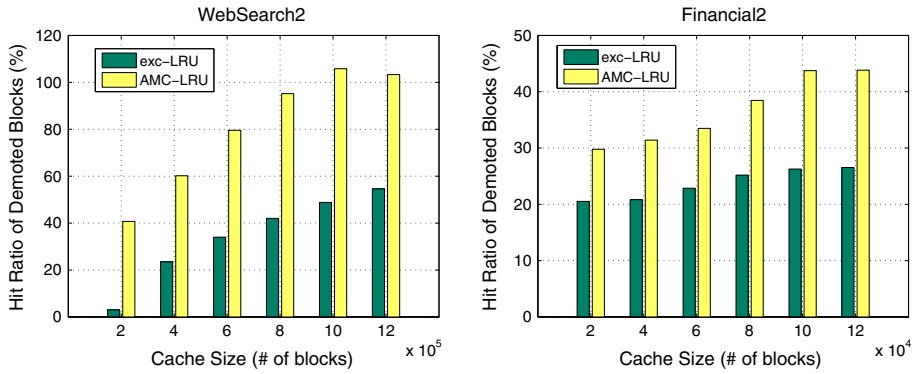
4.5. Selective promote/demote

From the DRAM cache hit ratio curve (Figure 8(b)), we have seen the selective promote operation of AMC-LRU effectively accumulates more hits in the DRAM cache than exc-LRU does. We now analyze how the selective demote operation helps improve the overall cache performance in the hybrid cache system.

DOI: 10.1002/cpe



(a) Number of Blocks Demoted into SSD cache



(b) Hit Ratio of Demoted Blocks

Figure 10. Analysis of the selective demote operation under the WebSearch2 and Financial2 traces: (a) number of blocks demoted into solid state drive (SSD) cache and (b) hit ratio of demoted blocks. AMC, adaptive multi-level cache; LRU, least recently used.

Figure 10(a) compares the number of blocks demoted into the SSD cache between exc-LRU and AMC-LRU under the WebSearch2 and Financial2 traces. The exc-LRU algorithm demotes every block evicted from the DRAM cache into the SSD cache. While the AMC-LRU algorithm takes advantage of the selective demote operation, it only demotes blocks into the SSD cache when the algorithm considers those blocks are worthwhile to keep in the cache. Therefore, as Figure 10(a) shows, the number of blocks demoted into the SSD cache in the AMC-LRU algorithm is significantly smaller than exc-LRU.

In order to analyze the effect of the selective demote operation, we tag the blocks demoted into the SSD cache and count the number of hits on these blocks. The hit ratio of demoted blocks is calculated using the number of hits on demoted blocks divided by the total number of demoted blocks. We observe the hit ratio of demoted blocks in AMC-LRU is much higher than exc-LRU, as shown in Figure 10(b). This demonstrates the selective demote operation can effectively filter out less useful blocks (relative to the blocks that reside in the LRU end of SSD cache) and demote more useful blocks into the SSD cache. The hit ratio of demoted blocks in AMC-LRU is even higher than 100% under a cache size of 10 and 12 ($\times 10^5$ blocks). This is because a demoted block may have more than one hit in the SSD cache.

4.6. Scalability

The proposed selective promote and demote operations and the adaptive probability adjustment policy used in the AMC-LRU algorithm are also suitable for other cache replacement algorithms, not just for the LRU algorithm, such as the ARC [28] algorithm. We implemented ind-ARC, exc-ARC, and AMC–ARC in the simulator. We initiated two independent ARC algorithms, L1 (DRAM

DOI: 10.1002/cpe

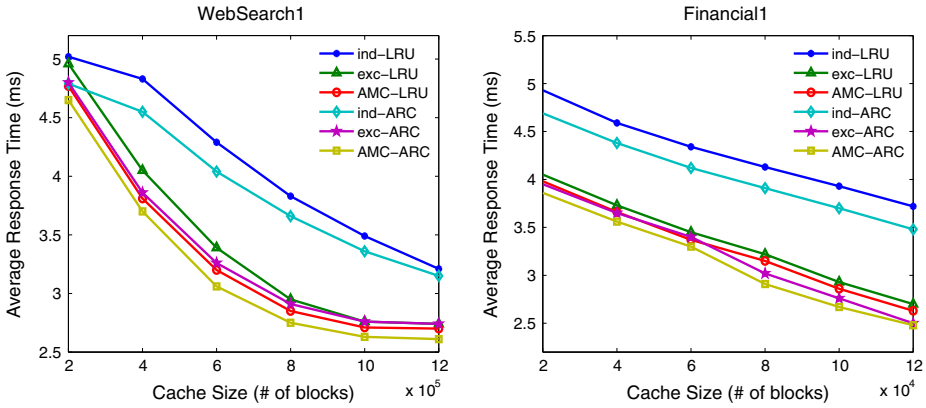


Figure 11. The average response time of six different algorithms. The adaptive multi-level cache (AMC) policy can be applied to both least recently used (LRU) and adaptive replacement cache (ARC) algorithms.

cache) ARC and L2 (SSD cache) ARC. The ind-ARC deploys the two independent ARC algorithms in DRAM and SSD caches, respectively. The exc-ARC implementation is similar to the exc-LRU algorithm, where new blocks are cached in DRAM using L1 ARC, blocks evicted from DRAM are cached in SSD using L2 ARC, and blocks hit in SSD are deleted from SSD and cached in DRAM using L1 ARC. The AMC–ARC is implemented by replacing the functions of do_LRU(L1,X) and do_LRU(L2,X) described in Section 3 with do_ARC(L1,X) and do_ARC(L2,X) (here, do_ARC() means the independent ARC algorithm). We do not alter the internal ARC algorithm.

Figure 11 presents the *avg_time* of different multi-level cache algorithms under the WebSearch1 and Financial1 trace. We find the performance improvement of AMC–ARC compared with ind-ARC and exc-ARC is similar to the performance improvement of AMC-LRU compared to ind-LRU and exc-LRU. This demonstrates our proposed AMC management policy is orthogonal to the design of single level cache replacement algorithms.

5. CONCLUSION

As SSD technology matures, more and more storage systems adopt hybrid cache hierarchies (DRAM–SSD–HDD). Effective management of multi-level hybrid cache systems becomes an important method to improve system performance. In this paper, we have discussed the design alternatives of multi-level, locally attached, hybrid cache management algorithms, and we have presented a novel multi-level hybrid cache algorithm that can adaptively keep 'hot' data blocks in DRAM and SSD caches. The proposed selective promote and demote operations effectively achieve cache exclusivity and also present better performance than the traditional multi-level cache algorithms. Our experimental results show that the AMC algorithm improves overall cache performance compared with ind-LRU and exc-LRU. The algorithm also extends SSD lifetime by reducing the total number of SSD write allocations.

ACKNOWLEDGEMENT

This work is supported by the National Science and Technology Major Project of the Ministry of Science and Technology of China under grant 2013ZX03003010-002.

REFERENCES

1. Flashcache. (Available from: https://github.com/facebook/flashcache) [accessed on 1 December 2014].
2. bcache. (Available from: https://http://bcache.evilpiepirate.org/) [accessed on 15 October 2014].
3. Canim M, Mihaila G, Bhattacharjee B, Ross K, Lang C. SSD bufferpool extensions for database systems,. *Proceedings of the 36th VLDB Conference*, Singapore, 2010; 1435–1446.

DOI: 10.1002/cpe

4. Klonatos Y, Makatos T, Marazakis M, Flouris MD, Bilas A. Azor: using two-level block selection to improve SSD-based I/O caches. *Proceedings of the 6th Internatioanl Conference on Networking, Architecture, and Storage*, Dalian, Liaoning, China, 2011; 309–318.
5. Wong TM, Wilkes J. My cache or yours? Making storage more exclusive. *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, Monterey, CA, USA, 2002; 161–175.
6. Chen Z, Zhang Y, Zhou Y, Scott H, Schiefer B. Empirical evaluation of multi-level buffer cache collaboration for storage systems. *SIGMETRICS Performance Evaluation Review* 2005; **33**(1):145–156.
7. Appuswamy R, Moolenbroek DC, Tanenbaum AS. Cache, cache everywhere, flushing all hits down the sink: on exclusivity in multilevel, hybrid caches. *Proceedings of the 29th International Conference on Mass Storage Systems and Technologies*, Long Beach, CA, USA, 2013; 1–14.
8. Gill B. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, San Jose, CA, USA, 2008; 4–4.
9. Jiang S, Zhang X. ULC: a file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. *Proceedings of the 24th International Conference on Distributed Computing Systems*, Hachioji, Tokyo, Japan, 2004; 168–177.
10. Yadgar G, Factor M, Li K, Schuster A. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Transactions on Computer Systems* 2011; **29**(2):1–51.
11. Chen Z, Zhou Y, Li K. Eviction-based cache placement for storage caches. *Proceedings of the USENIX Annual Technical Conference*, San Antonio, TX, USA, 2003; 268–282.
12. Grupp LM, Caulfield AM, Coburn J, Swanson S, Yaakobi E, Siegel PH, Wolf JK. Characterizing flash memory: anomalies, observations and applications. *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2009; 24–33.
13. Wei Q, Zeng L, Chen J, Chen C. A popularity-aware buffer management to improve buffer hit ratio and write sequentiality for solid-state drive. *IEEE Transactions on Magnetics* 2013; **49**(6):2786–2793.
14. Kgil T, Roberts D, Mudge T. Improving NAND flash based disk caches. *Proceedings of the 35th Annual International Symposium on Computer Architecture*, Beijing, China, 2008; 327–338.
15. Pritchett T, Thottethodi M. Sievestore: a highly-selective, ensemble-level disk cache for cost-performance. *Proceedings of the 37th Annual International Symposium on Computer Architecture*, Saint-Malo, France, 2010; 163–174.
16. Oh Y, Choi J, Lee D, Noh SH. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2012; 25–39.
17. Huang S, Wei Q, Chen J, Chen C, Feng D. Improving flash-based disk cache with lazy adaptive replacement. *Proceedings of the 29th International Conference on Mass Storage Systems and Technologies*, Long Beach, CA, USA, 2013; 1–10.
18. Guerra J, Pucha H, Glider J, Belluomini W, Rangaswami R. Cost effective storage using extent based dynamic tiering. *Proceedings of the 9th USENIX conference on File And Stroage Technologies*, San Jose, CA, USA, 2011; 273–286.
19. Chen F, Koufaty DA, Zhang X. Hystor: making the best use of solid state drives in high performance storage systems. *Proceedings of the International Conference on Supercomputing*, Tuscon, Arizona, USA, 2011; 22–32.
20. Appuswamy R, van Moolenbroek D, Tanenbaum A. Integrating flash-based SSDs into the storage stack. *Proceedings of the 28th International Conference on Mass Storage Systems and Technologies*, San Diego, CA, USA, 2012; 1–12.
21. Yang Q, Ren J. I-CASH: intelligently coupled array of SSD and HDD. *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, San Antonio, TX, USA, 2011; 278–289.
22. Badam A, Pai V. SSDAlloc: hybrid SSD/RAM memory management made easy. *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2011; 211–224.
23. Wang C, Vazhkudai SS, Ma X, Meng F, Kim Y, Engelmann C. NVMalloc: exposing an aggregate SSD store as a memory partition in extreme-scale machines. *Proceedings of the 26th International Parallel & Distributed Processing Symposium*, Shanghai, China, 2012; 957–968.
24. Kaiser J, Margaglia F, Brinkmann A. Extending SSD lifetime in database applications with page overwrites. *Proceedings of the 6th International Systems and Storage Conference*, Haifa, Israel, 2013; 11–11.
25. Johnson T, Jorge B. 2Q: a low overhead high performance buffer management replacement algorithm. *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994; 439–450.
26. Lee D, Choi J, Kim J, Noh SH, Min SL, Cho Y, Kim CS. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. *SIGMETRICS Performance Evaluation Review* 1999; **27**(1):134–143.
27. Jiang S, Zhang X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Performance Evaluation Review* 2002; **30**(1):31–42.
28. Megiddo N, Modha DS. ARC: a self-tuning, low overhead replacement cache. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, Berkeley, CA, 2003; 9–26.
29. Gill B, Modha D. SARC: sequential prefetching in adaptive replacement cache. *Proceedings of the USENIX Annual Technical Conference*, Anaheim, CA, USA, 2005; 293–308.
30. Gniady C, Butt A, Hu Y. Program counter based pattern classification in buffer caching. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, 2004; 1–27.
31. U. T. Repository. (Available from: http://traces.cs.umass.edu/index.php/storage/storage) [accessed on 1 October 2014].

DOI: 10.1002/cpe