FRD: A Filtering based Buffer Cache Algorithm that Considers both Frequency and Reuse Distance

Sejin Park
New Computing Lab
SK Telecom
baksejin@sk.com

Chanik Park
Dept. of Computer Science and Engineering
POSTECH
cipark@postech.ac.kr

***Abstract*—Buffer cache algorithms play a major role in filling the large performance gap between main memory and I/O devices in a mass storage system. Many buffer cache algorithms have been developed such as the low inter-reference recency set (LIRS) and adaptive replacement cache (ARC). Careful analysis of real-world workloads leads us to observe that approximately 50 to 90% blocks are accessed three or fewer times during the execution of various workloads. These infrequently accessed blocks are likely to cause high cache pollution by evicting better blocks from the cache. We also observe that these infrequently accessed blocks have certain access characteristics regarding reuse distance: either extremely long or short. Based on our observations, we propose an algorithm named frequency and reuse distance (FRD). The proposed algorithm concentrates on the manner in which to utilize both the access frequency and reuse distance of a block to determine the entries that must be stored in a buffer cache. We implemented the FRD algorithm using two LRU stacks. Experimental results show that the proposed algorithm outperforms state-of-the-art cache algorithms such as LIRS and ARC in most cases. We also show that FRD's hit ratio is stable under various cache sizes.**

Keywords—buffer cache; reuse distance; frequency

I. INTRODUCTION

Buffer cache algorithms play a major role in building a memory hierarchy in a mass storage system. For instance, a large performance gap between main memory and a hard disk drive (HDD) or solid state drive (SSD) is reduced by buffer cache. For network-attached storage, buffer cache is one of the key factors for improving performance. Because of its simplicity and low overhead, a least-recently-used (LRU) algorithm is one of the most commonly used buffer cache algorithms. However, the LRU algorithm performs poorly for the following workloads because of certain characteristics.

- Scanning workload. Because an LRU algorithm evicts the least-recently-used block, recently accessed blocks reside in the cache. However, the blocks in the scanning workload are accessed only a single time. In other words, the LRU algorithm generates considerable cache pollution. For instance, although meaningful blocks are present in the cache, the scanning workload evicts them.
- Cyclic access (loop-like) workload in which loop length is greater than cache size. For instance, when the cache size is 3 and the workload's block request sequence is 1-2-3-4-1-2-3-4-1-2-3-4, the LRU algorithm always generates a cache miss. In this case, Block 1 will be evicted as a result of the insertion of Block 4. Thus, the next request of Block 1 cannot be a cache hit. Therefore, if the cache size is smaller than the workload's cyclic pattern size, the LRU algorithm always generates a cache miss. This is an inevitable problem of the LRU algorithm.

To address the problems of LRU, many buffer cache algorithms have been proposed [1-10]. Among them, two cache algorithms, adaptive replacement cache (ARC) [2] and low inter-reference recency set (LIRS) [1], have shown the best performance for multiple workloads. The ARC algorithm consists of two queues: one for recency and the other for frequency. In the LIRS algorithm, cache entries are maintained in two LRU stacks in terms of inter-reference recency. These two-LRU stack-based approaches overcome the limitation of LRU algorithms. However, these sophisticated existing two-LRU stack-based approaches also suffer from noise blocks, that is, infrequently accessed blocks. Our observations, as described in Section 3, show that most real-world workloads have many noise blocks that are infrequently accessed and can pollute cache buffers in both ARC and LIRS algorithms. In an ARC algorithm, most frequently accessed blocks reside in a frequency queue. However, infrequently accessed blocks (e.g., only two or three times accessed blocks) also reside in the frequency queue because the algorithm determines the two or more accessed blocks are frequently accessed. The LIRS algorithm has the same problem. Two or more accessed blocks are considered low inter-reference recency blocks. Thus, they reside in the LIR stack even if they are infrequently accessed.

In this study, we propose a buffer cache algorithm, called frequency and reuse distance (FRD), that considers both frequency and reuse distance. Through careful analysis on various real-world workloads, we find that infrequently accessed blocks are dominant in most cases and such blocks are the main source of cache pollution. We concentrate on the manner in which to identify these infrequently accessed blocks and exclude them from the cache. The contributions of this study are as follows.

- We analyze real-world workload regarding buffer cache and find approximately 50 to 90% of blocks are infrequently accessed (three or fewer times) and their reuse distance is extremely long or short.

- We propose a simple but effective buffer cache algorithm that effectively filters out noise blocks that generate cache pollution.
- A two-LRU stack-based algorithm design can be easily implemented. It also offers O(1) time complexity for block eviction.

The remainder of the paper is organized as follows. Section 2 provides background to our study. A detailed analysis of real-world workload is presented in Section 3. Section 4 describes the proposed cache algorithm design and Section 5 presents several experimental results. Section 6 discusses the effectiveness of our proposed design and suggests future related studies. Section 7 reviews related work. Section 8 provides concluding remarks.

II. BACKGROUND

In this section, we briefly describe the state-of-the-art cache algorithms, LIRS [1] and ARC [2]. These two cache algorithms are generally known to outperform most cache algorithms [5-10] including LRU in practical workloads.

A. LIRS Algorithm

The LIRS algorithm is based on inter-reference recency (IRR), which is the same with reuse distance. Reuse distance represents the number of distinct blocks between two consecutive accesses to the same block in a request sequence. For example, when a request sequence is 3-1-2-4-0-2-3, the reuse distance of Block 3 is 4 and the reuse distance of Block 2 is 2. Therefore, each block has its own IRR (reuse distance) at time t. The basic idea of the LIRS algorithm is to maintain blocks with smaller IRRs in a cache. When a new block arrives, the LIRS algorithm does estimate the new block's IRR and decides whether a resident block is to be evicted based on the estimated IRR. To estimate the new block's IRR, the LIRS algorithm maintains the access history information. When the IRR of a new block cannot be estimated because of a lack of history information (e.g., when the block was first accessed), the LIRS algorithm stores the newly arrived block into a temporary cache buffer that is isolated from the main cache buffer in which all blocks are sorted according to their IRR. This is why LIRS algorithm uses two cache buffers in its design; an HIR stack (or temporary cache buffer) and an LIR stack (or main cache buffer).

The LIRS algorithm classifies each block into two categories according to IRR: HIR for high inter-reference recency and LIR for low inter-reference recency. If an incoming block is accessed for the first time, (i.e., when a cache miss occurs and no access history information is available), then the incoming block enters the HIR stack and the associated access history information is inserted into the LIR stack. Otherwise, if the incoming block is accessed again (i.e., when a cache hit or miss occurs but access history information is available), the incoming block is moved to an LIR stack. This is because the IRR value of the incoming block is smaller than that of other blocks in the LIR stack. The LIRS algorithm adopts a heuristic strategy such that when a block is evicted from an LIR stack, it is reinserted back into an HIR stack. Therefore, cache eviction occurs only in an HIR stack. However, a limitation exists in the LIRS algorithm whereby the access frequency of blocks in an LIR or HIR stack is not to be considered. In other words, if an incoming block is found in an LIR or HIR stack (a cache hit), it is inserted into the top of the LIR stack regardless of its access frequency. This decision increases cache pollution for infrequently accessed blocks.

B. ARC Algorithms

To manage both recency and frequency in block accesses, the ARC algorithm uses two LRU cache buffers, the sizes of which are dynamically adjusted. In this algorithm, a new block enters into the first LRU stack. This LRU stack is labeled T1 for recency. If the block is accessed again (i.e., a cache hit or miss occurs by a history buffer hit), the block is moved to the second LRU stack labeled T2 for frequency. T1 and T2 have their own history buffers named B1 and B2, respectively. In these history buffers, the block's metadata are maintained when a block is evicted from T1 or T2. The sizes of both T1 + T2 and B1 + B2 are the same as that of the cache. Interestingly, the ARC algorithm has a unique feature that adaptively balances the workload's recency and frequency. If the recency is dominant during some execution interval of a workload, T1 increases. If the frequency is dominant during another execution interval of a workload, T2 increases. However, some limitations exist with the ARC algorithm. First, infrequently accessed blocks can be maintained in T2 if the blocks are accessed more than twice. This causes T2 cache to be polluted. Second, supporting blocks with long reuse distance is difficult because its history buffer size is limited to cache size.

C. Brief Summary of the Two Algorithms

These two state-of-the-art cache algorithms have many similarities, even though their hit ratio results are quite different.

- First, these two algorithms consist of two LRU stacks to separate workloads according to their policies. They also maintain history information for the purposes of future block access.
- One major purpose of the first stack of the algorithms is to make cache is scan-resistant. A newly requested block is entered into the first stack and, if it is reused or its history is accessed, it can enter into the second stack. Thus, the second stack can maintain meaningful blocks.
- In most cases, these two algorithms outperform the LRU algorithm.

Although the two algorithms ensure the cache is scan-resistant, they possess limitations in terms of handling infrequently accessed blocks. According to our analysis on real-world workloads, most blocks are infrequently accessed and these blocks can pollute the second stack that contains meaningful blocks. The LIRS algorithm does not consider access frequency. Thus, infrequently accessed blocks can cause LIR stack pollution. The ARC algorithm also has the same pollution problem and it does not consider blocks that have a long reuse distance. It only concentrates on blocks having a short reuse distance.

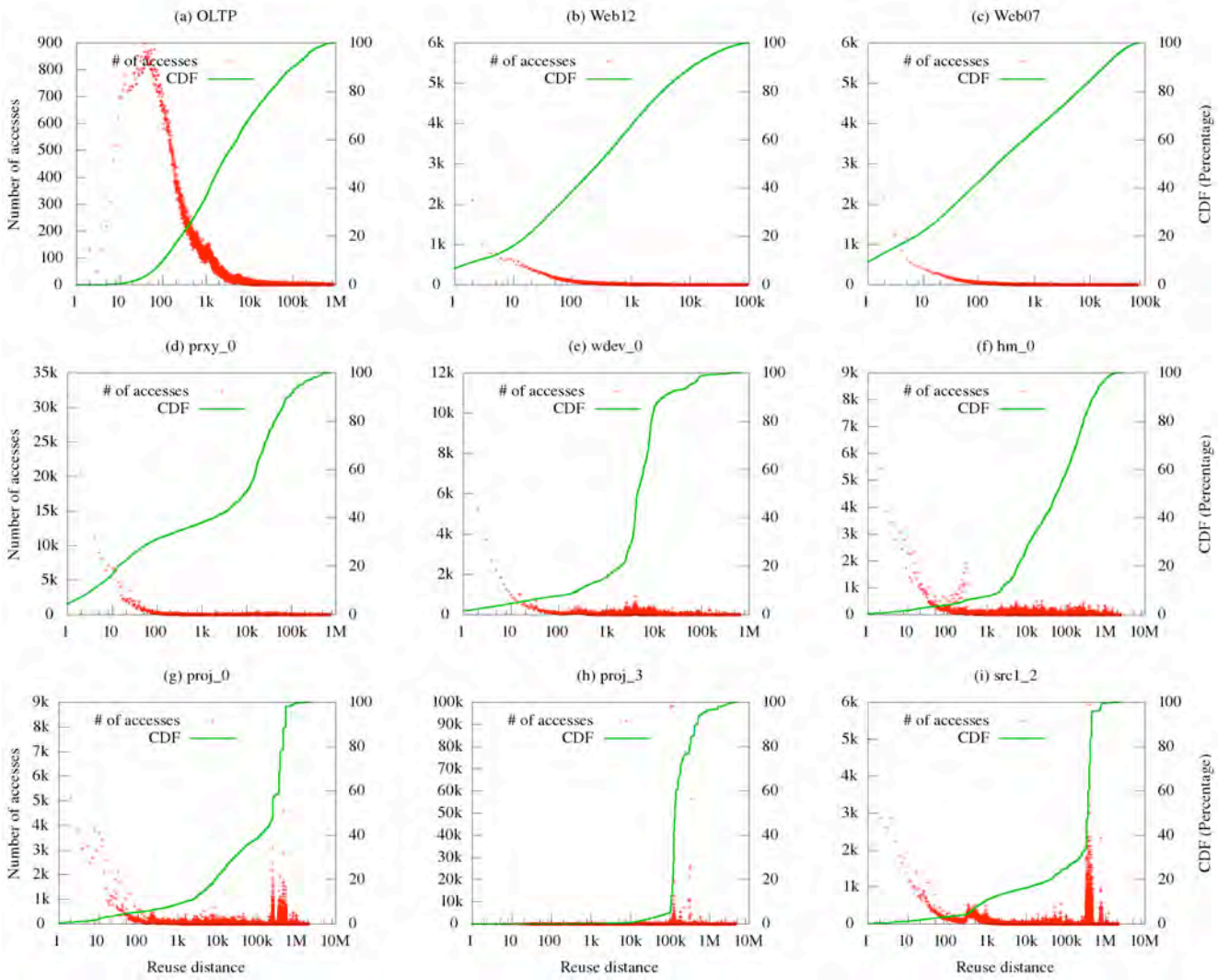

Figure 1. Reuse distance distribution of each workload. The x axis uses a logarithmic scale. Each workload has its dominant reuse distance period. In the case of workload (h) proj_3, blocks with a 100 K to 1 M reuse distance period are dominant (more than 95% of the entire workload fall within this period).

Table 1. Workload description

Name	Type	Description
OLTP	Application	Online transaction processing [1]
Web12	Web server	A typical retail shop [1]
Web07	Web server	A typical retail shop [1]
prxy_0	Data center	Firewall/web proxy [2]
wdev_0	Data center	Test web server [2]
hm_0	Data center	Hardware monitoring [2]
proj_0	Data center	Project directories [2]
proj_3	Data center	Project directories [2]
src1_2	Data center	Source control [2]

III. Workload Analysis and Observations

In this section, we analyze a wide range of real-world workload sets from [11, 12] collected from a single application trace to various server or working directory traces in data centers.

Table 1 lists the characteristics of each workload. The OLTP workload contains a CODASYL database related to a one-hour period and was used in [2, 4, 5, 9]. The Web12 and Web07 workloads are both web server workloads that contain access traces of detail pages about events (music, theater, etc.) in Munich. These two sets of traces were recorded in December 2012 and July 2013, respectively. The traces are long-tail distributions which may be typical for retail shops [11]. from the workloads prxy_0 to src1_2 are block-level traces for a single week in an enterprise data center from MSR Cambridge [12]. We chose 9 workloads from whole sets in [11, 12] neither too large nor too small working sets.

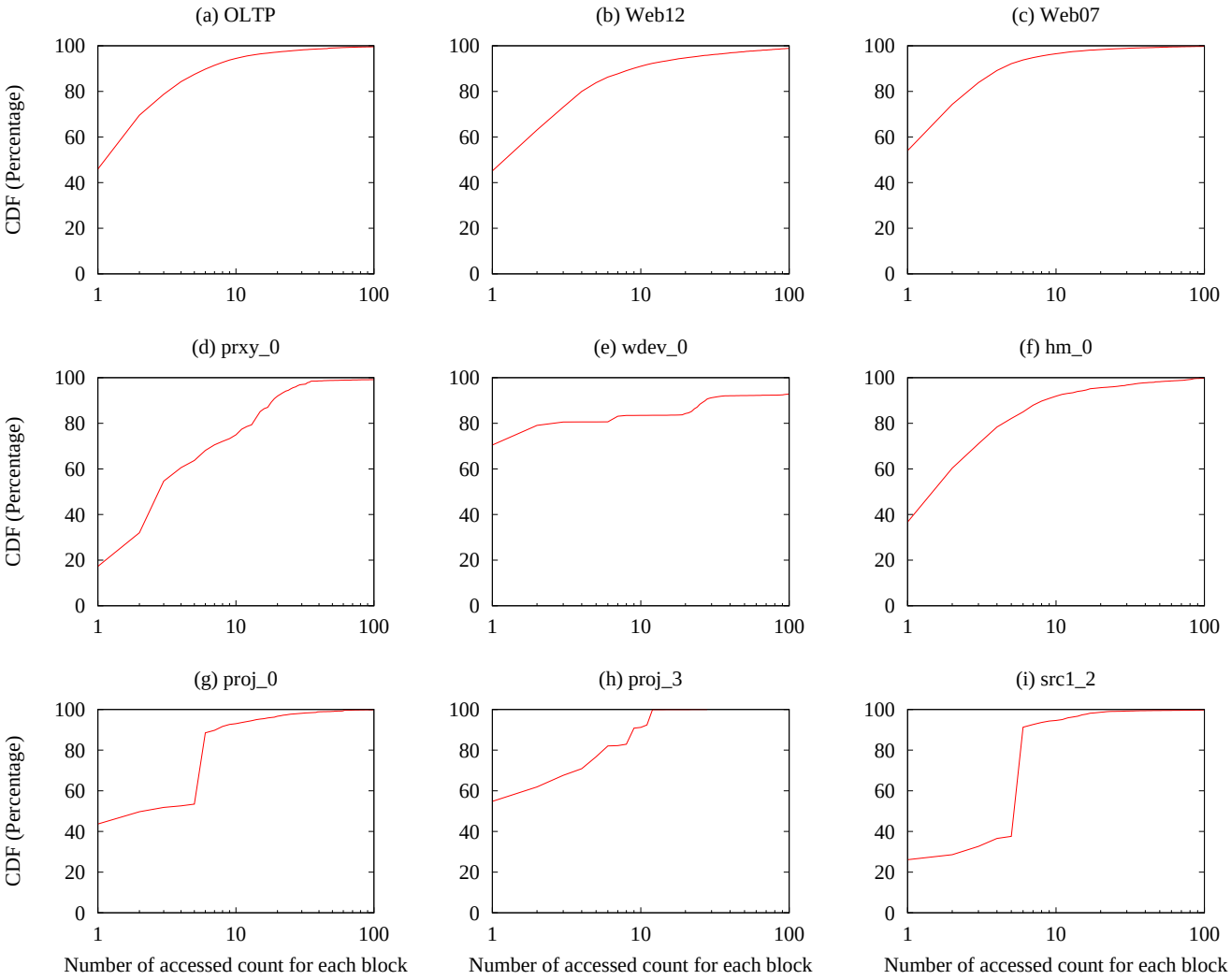

Figure 2. The cumulative distribution function (CDF) for the number of accessed counts for each block. The x axis uses a logarithmic scale. In most cases, blocks that are accessed three or fewer times are approximately 50-90% of the CDF. This kind of block is likely to be a noise that pollutes cache.

Figure 1 shows the reuse distance distribution of each workload. Specifically, it shows that each workload has a dominant reuse distance period. In the case of (h) proj_3, more than 95% of the workload's reuse distance is 100 K to 1 M. For a 4-KB block size, this period is approximately 409.6–4096 MB in cache size. Thus, if a cache size is less than 400 MB, it shows an extremely low hit ratio for an LRU or LRU-like algorithm (see Figure 5 in Section 5, "Evaluation"). A dominant reuse distance period may be a good candidate for designing an effective cache algorithm, but it depends on workload. As depicted in Figure 1, a dominant reuse distance does not have any common period among various workloads. To explore each block's access characteristics, we analyze the number of accesses of each block. Figure 2 shows the cumulative distribution function (CDF) for the number of accesses of each block. Note that approximately 50 to 90% of all blocks are accessed three or fewer times in most cases. This is our first observation for designing a new cache algorithm.

Observation #1. Approximately 50 to 90% of blocks are infrequently accessed in a real-world workload.

Observation #1 means that most blocks are infrequently accessed and only a few blocks are frequently accessed. We expect that if a cache algorithm performs in an LRU manner, blocks to be frequently accessed may be easily evicted from a cache because of infrequently accessed blocks. Therefore, when a cache is full, infrequently accessed blocks should be evicted prior to frequently accessed blocks, and frequently accessed blocks should be retained for a long time to make more cache hits.

For more detailed observation on the characteristics of infrequently accessed blocks, we analyze the reuse distance distribution of infrequently accessed blocks. Figure 3 shows the reuse distance distribution for infrequently accessed blocks (e.g., blocks accessed three or fewer times). Here, the reuse distance is represented by the percentage of a given cache size. For instance, if the reuse distance is 1024 blocks, 4 MB is the reuse distance in size in which block size is 4 KB. Thus, if a

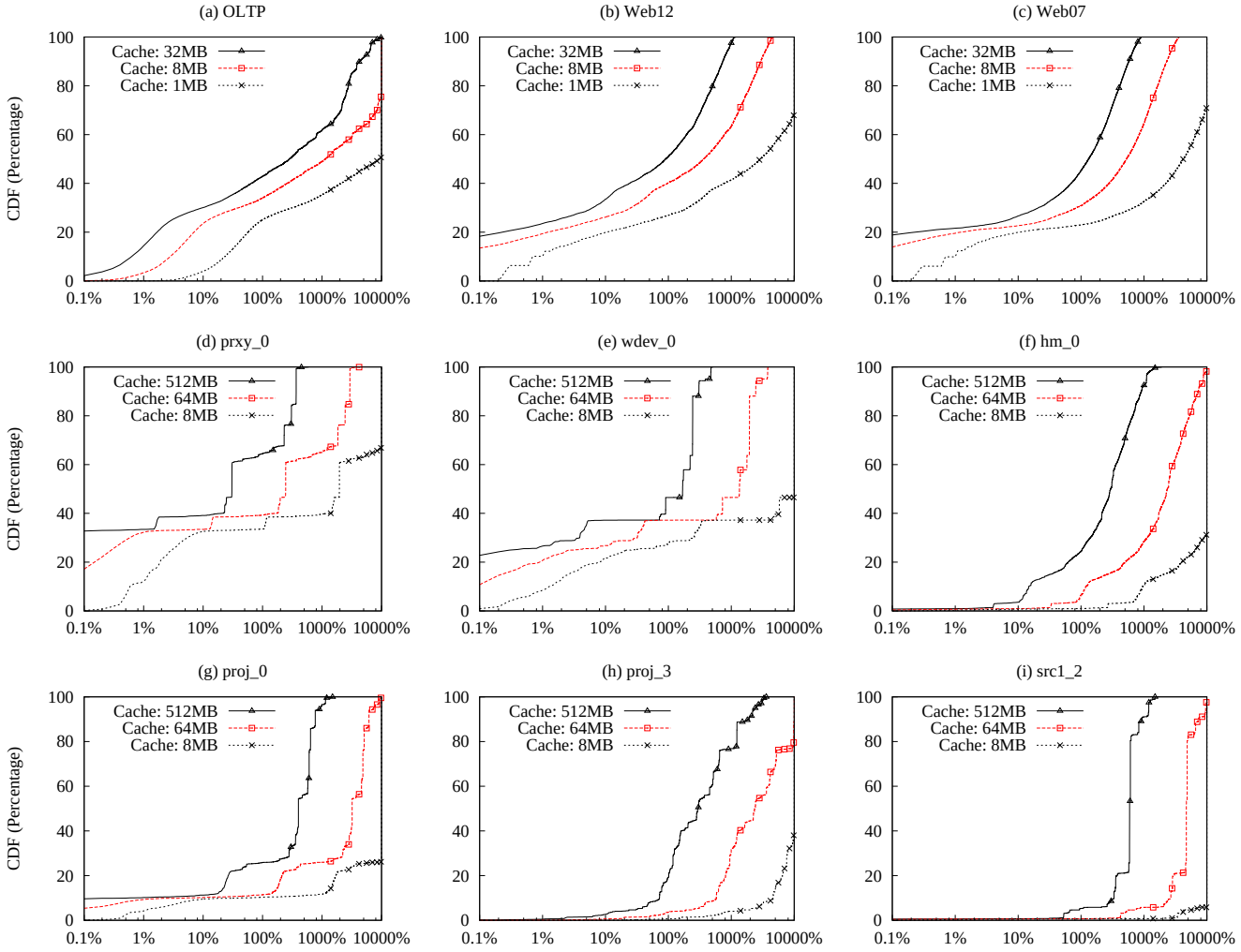

Figure 3. Reuse distance for three or fewer times in which blocks are accessed. The reuse distance is represented by the percentage of cache size. Because the working set size of each workload is different, we analyze according to various cache sizes (1, 8 and 32 MB or 8, 64, and 512 MB). The x axis uses a logarithmic scale. A reuse distance of 100% means the reuse distance is the same as the cache size, and 1000% means 10 times the cache size.

cache size is 8 MB, the reuse distance of 1024 is 50% the cache size. We analyze a reuse distance distribution using various cache sizes. From Figure 3 (a) to (i), we can classify these infrequently accessed workloads into two types according to reuse distance distribution:

- Type 1: Infrequently accessed with a long reuse distance.
- Type 2: Infrequently accessed with a short reuse distance.

In the LRU algorithm, Type 1 blocks generate pure cache pollution and this algorithm cannot be a cache hit as well. Apparently, we must exclude these kinds of blocks from a cache buffer. The Type 2 blocks might be a cache hit because it has a short reuse distance. However, after a cache hit, it still produces cache pollution because it has infrequently accessed characteristics. In Figure 3, Workloads (f)–(i) can be classified as Type 1 and Workloads (a)–(e) can be classified as a combination of Types 1 and 2. Interestingly, in all workloads, reuse distance distribution from 10 to 100% of a given cache size is extremely low. For instance, in the Workload (d) prxy_0 with 8-MB cache size, the distribution from 10 to 100% of cache size proves to be less than 1%. Other workloads show a similar trend. In other words, most infrequently accessed blocks are distributed in the range of 0 to 10% of a given cache size or greater than 100% of a given cache size. This is our second observation.

Observation #2. Reuse distance for infrequently accessed blocks is either very short or very long. Regarding cache size, most distribution is less than 10% or greater than 100% of cache size.

IV. Proposed Algorithm

In this section, we propose a buffer cache algorithm that considers both access frequency and reuse distance, known as FRD. This algorithm is based on the two observations described in Section 3. The FRD effectively filters out cache polluting blocks using a filter stack and effectively maintains meaningful blocks using a reuse distance stack.

A. Block Classification

From Observation #1, infrequently accessed blocks are approximately 50 to 90% of all blocks. Thus, we must exclude these infrequently accessed blocks. In Section 3, we have classified the infrequently accessed blocks by reuse distance (Type 1 and 2). In this section, we expand the block classification into four classes. We classify characteristics of blocks by combining frequency and reuse distance. Table 2 shows the block classification.

Table 2. Block Classification

Class	Description
FS	Frequently accessed, short reuse distance
FL	Frequently accessed, long reuse distance
IS	Infrequently accessed, short reuse distance
IL	Infrequently accessed, long reuse distance

Class FS and FL blocks are the most preferable blocks with respect to the cache algorithm because these blocks are likely to generate cache hits. By contrast, Class IS or IL blocks are likely to pollute a cache in most cases. However, Class IS can also generate a cache hit because Class IS blocks can be reused (e.g., a cache hit by its short reuse distance). However, immediately after reuse, they begin to pollute a cache. Therefore, these kinds of blocks should be carefully handled in a cache buffer. Of course, Class IL blocks should not be maintained in a cache.

B. Two-buffer-based Design

Cache pollution caused by Class IS or IL commonly occurs as a result of a scan-like workload in a LRU cache algorithm. To avoid this cache pollution, we design our cache algorithm using two types of buffers: temporal and actual cache buffers. In this design, a newly requested block is inserted into a temporal cache buffer. If any block in the temporal buffer is re-accessed, then this block is moved to the most recently used (MRU) position of the temporal buffer (Class FS or IS). If a block in the temporal buffer is not re-accessed in a short time, then this block is evicted because the temporal buffer size is relatively small. This simple design effectively causes a cache algorithm to be scan-resistant. The proposed algorithm also retains block accessed history as a history block so that the evicted block can be inserted into the actual cache buffer. If the evicted block is re-accessed, it moves to the MRU position of the actual cache buffer because it is considered a frequently accessed block with long reuse distance (Class FL). Therefore, the dominant Class FL blocks are maintained in the actual cache buffer. In addition, Class IS or IL cannot produce cache pollution because those blocks are located in the temporal buffer and quickly evicted without any interference to the actual cache buffer.

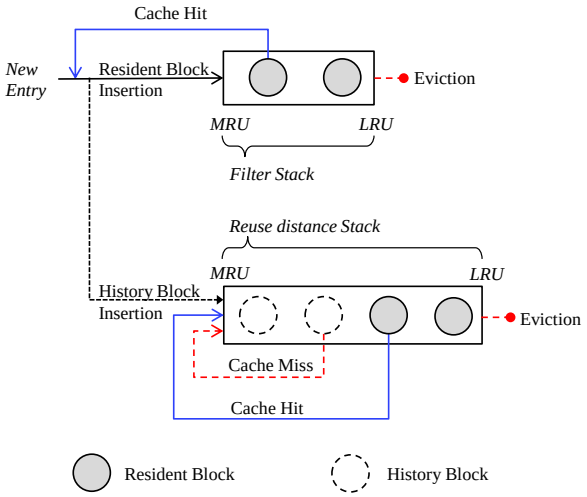

Figure 4. Illustration of FRD cache algorithm. Filter stack and reuse distance stack are managed in an LRU manner. When a new block is requested, its resident block is inserted into the filter stack and its history block is inserted to the re use distance stack.

In our proposed cache algorithm, two types of blocks are maintained: (1) a history block that does not have actual data but only metadata (e.g., block number); (2) a resident block that has actual data. Figure 4 is an illustration of our proposed algorithm (FRD) that maintains two cache buffers. The temporal buffer and the actual buffers are named filter stack and reuse distance stack, respectively.

The filter stack has two purposes. One purpose is to identify noise blocks (e.g., Class IS or IL), and the other purpose is to identify blocks with short reuse distance (e.g., Class IS or FS). The reuse distance stack also has two purposes. One purpose is to maintain history block in reuse distance order up to the last resident block in the reuse distance stack. The other purpose is to store frequently accessed blocks for later used (e.g., Class FL).

The filter stack contains only resident blocks, but the reuse distance stack contains both resident and history blocks. When a block is inserted into the filter stack, its history block is automatically generated and inserted into the reuse distance stack. Thus, regardless of block types (resident or history), the reuse distance stack is ordered with a reuse distance sequence.

The reuse distance stack maintains history blocks up to the oldest resident block. In other words, the maximum reuse distance for the history block in this algorithm is at most the oldest resident block's reuse distance minus 1. As depicted in Figure 4, each stack is managed in an LRU manner and has its own eviction point at the end of the stack (e.g., the LRU position of each stack).

Initially, both the filter and reuse distance stacks are filled with newly arrived blocks from the reuse distance stack to the filter stack. Therefore, we can assume without loss of generality that both filter and reuse distance stacks are fully occupied. By default, the filter stack size is 10% of the given cache size based on our Observation #2.

When the two stacks are full, a total of four cases are possible for a block request because of the history block accesses. Specifically, a cache miss has two cases: one with and one without a history hit. A cache hit also has two cases: one in the filter stack and one in the reuse distance stack.

Detailed descriptions of each case are as follows.

- **Case 1) Cache miss and history miss:** Evict the oldest block in the filter stack. Then, insert the missed block into the filter stack and generate a history block for the missed block. In addition, insert the history block into the reuse distance stack. No eviction occurs in the reuse distance stack because the history block contains only metadata.
- **Case 2) Cache miss but history hit:** Remove all history blocks between the 2nd oldest and the oldest resident blocks. Next, evict the oldest resident block from the reuse distance stack. Then, move the history hit block to the MRU position in the reuse distance stack and change it to a resident block. No insertion or eviction occurs in the filter stack.
- **Case 3) Cache hit in the filter stack:** Move the corresponding block to the MRU position of the filter stack. The associated history block should be updated to maintain reuse distance order (i.e., move its history block in the reuse distance stack to the MRU position of the reuse distance stack).
- **Case 4) Cache hit in the reuse distance stack:** Move the corresponding block to the MRU position of the reuse distance stack. If the corresponding block is in the LRU position of the reuse distance stack (i.e., the oldest resident block), the history blocks between the LRU position and the 2nd oldest resident block are removed. Otherwise, no history block removing occurs.

C. Analysis of FRD Algorithm

In the proposed algorithm, the size of the filter stack is the only tunable parameter. Based on Observation #2, we allocate 10% of the cache size for the filter stack. Note that most of the infrequently accessed blocks have either very short or very long reuse distance and most of their distributions are less than 10% or greater than 100% of the cache size. Although we set the filter stack size to 10% of the total cache size by default, the experimental results in Section 5 reveal that the efficiency of the proposed algorithm is insensitive to this parameter.

We next describe the manner in which the proposed algorithm handles the four block access types shown in Table 2.

If a block is Class FS, it will reside in the filter stack with frequent cache hits. If the filter stack is not sufficiently large to cache it, a cache miss will occur in the filter stack. However, the block will be moved to the reuse distance stack because of its short reuse distance. Once it is stored in the reuse distance stack, the block access contributes to a cache hit.

If a block is Class FL, it may generate a cache miss at the second request because the filter stack size is not sufficiently large to cover the block's reuse distance. However, the block will be maintained in the reuse distance stack because the block's reuse distance is most likely to be shorter than the longest reuse distance. From this point forward, the block access will contribute to many cache hits for a considerable period.

If a block is Class IS, the block access is most likely frequent in some intervals and infrequent in other intervals. In the proposed algorithm, the block is likely to be maintained in the filter stack during frequent access intervals and will be evicted from the filter stack during infrequent access intervals. Therefore, the proposed algorithm not only generates cache hits from the blocks in Class IS but also effectively isolates them from any blocks in other classes (e.g., Classes FS and FL).

If a block is Class IL, the block is always inserted into the filter stack because the associated history information cannot be found because of its large reuse distance. Therefore, these infrequently accessed blocks cannot pollute the reuse distance stack. Note that the filter stack effectively filters out Class IS and IL blocks because not having those blocks inserted into the reuse distance stack is preferable.

Because the proposed algorithm is based on two LRU stacks, cache eviction is accomplished with O(1) time complexity.

D. Comparison of FRD to ARC to LIRS

Similar to ARC and LIRS, FRD uses two cache buffers in its design. ARC's T1, LIRS's HIR stack, and FRD's filter stack are the first stack; and ARC's T2, LIRS's LIR stack, and FRD's reuse distance are the second stack. One primary purpose of the first stack is to make the cache scan-resistant. In other words, one-time accessed blocks cannot enter into the second stack.

However, the manner in which to handle the first stack in FRD is different from the operations in ARC and LIRS. When an incoming block is found in the first stack, that is, a cache hit occurs there, FRD moves the block to the top of the first stack. By contrast, LIRS and ARC move the hit block to the top of the second stack. This decision in both LIRS and ARC cannot prevent such blocks from causing cache pollution in the second stack.

FRD has two eviction points. These are the LRU positions in the two stacks. This causes complete isolation of the blocks having similar characteristics. However, LIRS has only a single eviction point: the LRU position of the HIR stack.

FRD maintains all history blocks up to the oldest resident block in the second stack. By contrast, the ARC maintains history up to cache size * 2. This limitation of the ARC algorithm causes poor performance for workloads whose reuse distance is greater than a given cache size.

V. EVALUATION

To show the performance of our FRD algorithm, we implemented a trace driven simulator written in C. The simulator contains one fixed-size LRU stack for the filter stack and one resizable LRU stack for the reuse distance stack to maintain history blocks dynamically. In this evaluation, we did not count the metadata size for the cache because it is negligible. Detailed information about the overhead is explained in Section 5.5

We compared the proposed FRD algorithm with the LRU, LIRS, ARC, and OPT [13] algorithms using various real-world workloads listed in Table 1. The OPT is an optimal offline cache algorithm that is not feasible as online cache. However, it is useful for comparing the maximum performance with that of various cache algorithms. For LIRS algorithm, we set 1% of the cache size and unlimited LIR stack size for the non-resident block as the default setting from the LIRS algorithm [1].

A. Workload Hit Ratio Analysis

Figure 5 shows the experimental results for real-world workloads. On average, FRD showed the highest hit ratio for various cache sizes among the cache algorithms. We precisely analyzed the results of the hit ratio comparison for each algorithm.

(a) OLTP: As previously determined in [2], the ARC algorithm outperforms LIRS in this workload. However, in our study, FRD outperformed ARC. This is because ARC could not effectively consider those blocks having a long reuse distance. In this workload, more than 70% of the blocks were infrequently accessed (Figure 2) and the short reuse distance accesses were dominant (Figure 1). We could then conclude that this workload is Class IS dominant (i.e., is infrequently accessed and has a short reuse distance). As explained, in the LIRS algorithm, the Class IS block can easily pollute the second stack (LIR stack). In addition, many cache misses occur because of a relatively small first stack size (HIR stack).

(b) Web12 and (c) Web07: These workloads have many blocks of Classes IS and IL (Figure 3) and more than 80% blocks are infrequently accessed (Figure 2). Thus, LIRS suffers from cache pollution by blocks of Class IS; ARC shows better performance than LIRS. However, on average FRD performed better than ARC because ARC does not handle blocks with a long reuse distance even though they are frequently accessed.

(d) prxy_0: This workload has many Class IS blocks (Figure 3) and the results show that FRD performed best for all cache sizes. In other words, FRD not only generated many cache hits, but also effectively filtered out Class IS blocks. Other algorithms cannot filter out Class IS blocks, which causes cache pollution. This is why their hit ratios are low.

(e) wdev_0: LIRS and FRD showed similar hit ratio curves, but FRD performed better. When cache size was small (≤ 4 MB), ARC performed the best, but when cache size was large (> 4 MB), ARC’s performance was worse than that of both LIRS and FRD. This derives from the reuse distance distribution of wdev_0 (Figure 1). This workload contains more blocks having long reuse distances than short reuse distance.

(f) hm_0: In this workload, blocks of Class IL are dominant, whereas blocks of Class IS are small (Figure 3). Thus, algorithms such as LIRS and ARC do not suffer from cache pollution with this kind of workload. The reuse distance distribution of this workload also has no special point or period (Figure 1). As depicted in Figure 5, it showed moderate results for all cache algorithms. Figure 5 does not reveal the algorithm that performed best, but Table 3 reveals that the overall average performance for the FRD algorithm remained the highest .

(g) proj_0: In most cases, FRD outperformed ARC and LIRS. As depicted in Figure 3, this workload contains approximately 10% of Class IS blocks that cause cache pollution. The results show that FRD effectively filtered out these blocks. Interestingly, the ARC algorithm shows relatively worse hit ratio than ARD or FRD when cache size is 512 MB. Figure 1 reveals the reason for this limitation. More than 50% of the reuse distance distribution is concentrated in the period from 100 K to 1 M. In a 4-KB block system, this value is converted to 409.6 to 4096 MB size. Because the ARC algorithm maintains history blocks at most cache size * 2, the blocks having a long reuse distance (e.g., longer than cache size * 2) cannot enter into the frequency stack and this generates many cache misses.

(h) proj_3: On average, LIRS showed the best performance for this workload. The filtering effect was minimized because most infrequently accessed blocks have a long reuse distance (Class IL). In fact, this workload is a special case. In Figure 1, the reuse distance distribution for proj_3 shows that more than 95% of blocks are concentrated in the period from 100 K to 1 M. Structurally, the LIRS algorithm can maintain longer history blocks than can the FRD algorithm because the oldest resident block in the LIR stack is moved to the HIR stack. This is because the LRU position of the HIR stack is the only eviction point. This is the reason the LIRS performs best for this workload. By contrast, the FRD algorithm does not move the oldest resident block in the reuse distance stack to the filter stack but instead evicts it. In FRD, the filter stack has meaningful blocks such as Class IS or FS. Therefore, moving the oldest resident block to the filter stack generates more cache pollution for the filter stack. Note that the HIR stack of the LIRS algorithm is used as a temporal space that contains a mixed set of new and old blocks. There’s one more interesting point at 1024MB cache size. ARC shows the best hit ratio. This is because of the LRU-friendly workloads. The relative hit ratio difference of LRU compared to LIRS or FRD at 1024 MB is smaller than other cache sizes. It means, there are many blocks at 1024 MB cache size environment that LRU algorithm can handle. Specifically, LIRS or FRD can generate inevitable cache miss at second block access whose reuse distance is longer than the HIR size or filter stack size, respectively. However, LRU can generate cache hit at second block access for those blocks and ARC can adjust its T1 stack that works like LRU. Therefore, because LIRS or FRD generates more cache miss than ARC in this situation, ARC shows the best performance.

(i) src1_2: If cache size was small (≤ 64 MB), ARC performed the best, but if cache size was large (> 64 MB), LIRS performed the best. However, on average, FRD shows the most stable result which is revealed in Table 3. Like (h)

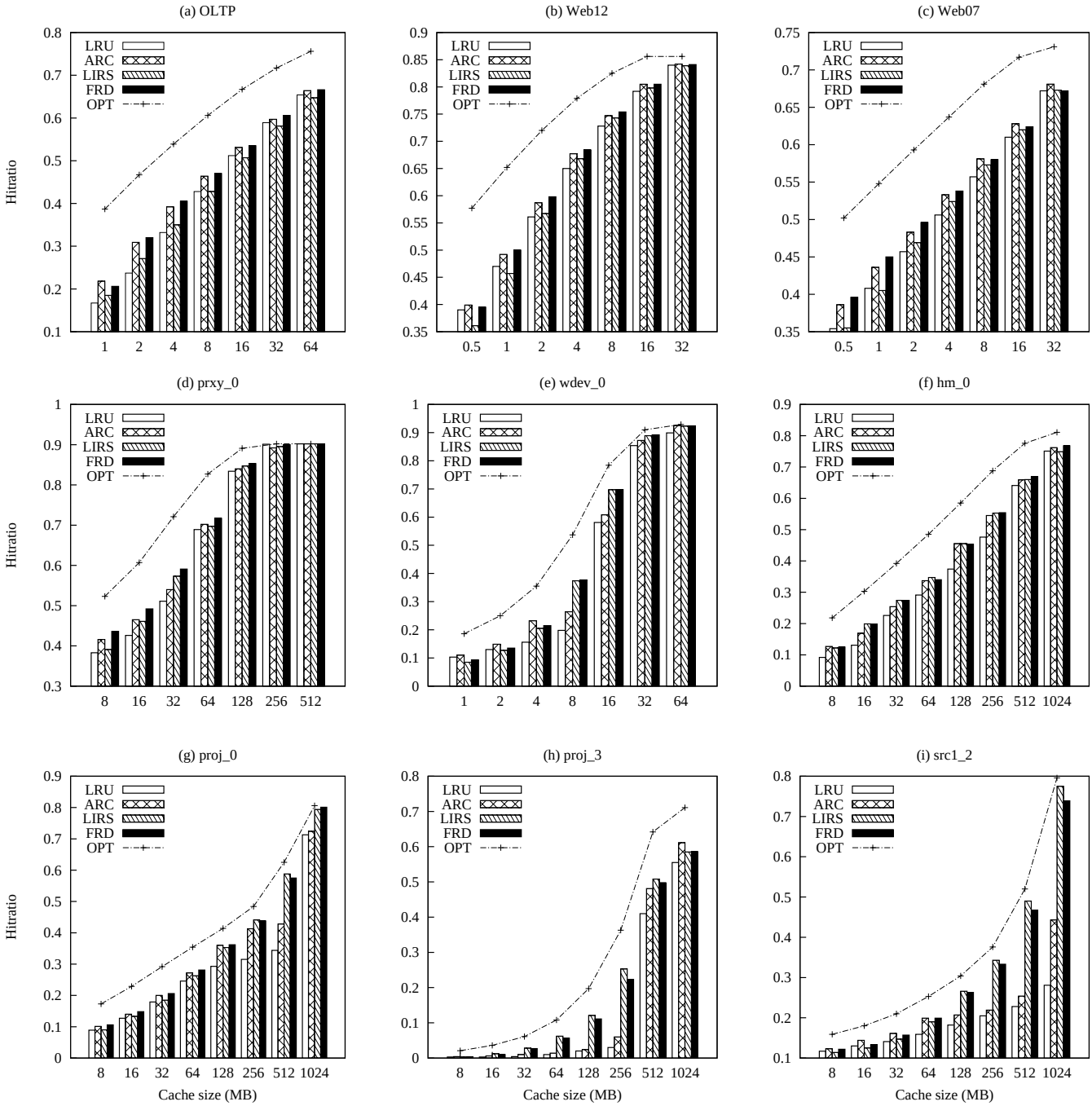

Figure 5. Hit ratio comparison using LRU, ARC, LIRS, OPT and FRD cache algorithm.

proj_0, more than 50% of the reuse distance distribution is concentrated in the period from 100k to 1M. However, (i) src1_2 has one more concentrated point around the period from 100 to 1k (Figure 1) and it contains rare Class IS blocks (Figure 3). That means, a cache algorithm does not suffer from cache pollution by Class IS blocks and it results in good performance in ARC when the cache size is small and good performance in LIRS when the cache size is large. Note that, when cache size is small, ARC could maintain newly accessed blocks longer than LIRS because ARC's T1 could bigger than LIRS' HIR stack size.

B. Cache Performance Stability

In our evaluation, the FRD algorithm showed the most stable performance with respect to various cache sizes. This means that FRD does not have a significantly low hit ratio for specific cache sizes. However, other algorithms showed a similar unstable point. The LIRS algorithm performs poorly for: (a) OLTP, (b) Web12, and (c) Web07 when cache size is small. This is because of Class IS blocks. As presented in Figure 2 and 3, these workloads have many blocks of Class IS. The blocks of Class IS easily pollute the LIR stack with the

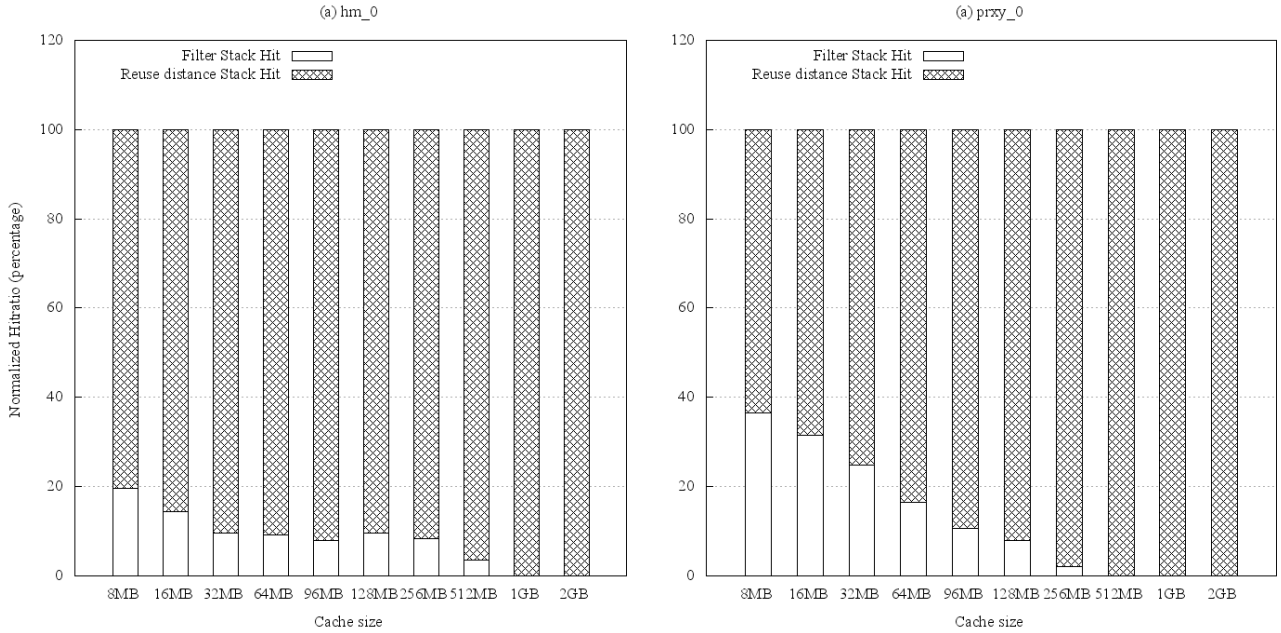

Figure 6. Hit ratio analysis of the filter stack and reuse distance stack.

LIRS algorithm. This results in a low hit ratio when cache size is small. However, in the case of FRD, the cache performance is more stable than that of LIRS even if the cache size is small. This result derives from the filter stack, which does not allow the Class IS block to be inserted into the reuse distance stack.

The ARC and LRU algorithms perform poorly for: (g) proj_0 and (h) proj_3 when cache size is small. This is because of the Class FL blocks. As depicted in Figure 2 and 3, these workloads have many blocks of Class FL. Even if a block is frequently accessed, its reuse distance is greater than the cache size. This means it is difficult for cache hits to occur with the ARC and LRU algorithms. This is because of the limited history stack size. In the case of (h) proj_3, the ARC hit ratio for 256 MB is approximately three times less than that of the LIRS or FRD algorithm. This is a major problem because cache size can be smaller than the working set size. However, the FRD can support blocks of Class FL because of the reuse distance stack with history blocks.

C. Overall Comparison with OPT

Although FRD shows the best performance among existing cache algorithms, a performance gap exists with OPT. Figure 5 contains the hit ratio comparison between FRD and OPT.

Table 3 summarizes the overall results when using an average hit ratio value. We calculate the "hit ratio of a given cache algorithm / hit ratio of OPT algorithm" for all cache sizes and we calculate their average values. As shown in Table 3, FRD outperforms other state-of-the-art cache algorithms except for workload proj_3, which has a special reuse distance distribution. As revealed, designing a better cache algorithm is possible.

Table 3. Overall result: an average value of each algorithm's hit ratio over that of OPT. Close to 1.0 means approximating OPT's hit ratio. (Underscored: best value)

	LRU	ARC	LIRS	FRD
OLTP	0.674	0.746	0.691	***0.753***
Web12	0.829	0.852	0.827	***0.857***
Web07	0.800	0.839	0.812	***0.847***
prxy_0	0.844	0.870	0.870	***0.898***
wdev_0	0.647	0.723	0.728	***0.745***
hm_0	0.598	0.700	0.723	***0.724***
proj_0	0.612	0.722	0.740	***0.780***
proj_3	0.172	0.241	***0.516***	0.478
src1_2	0.620	0.697	0.799	***0.813***

D. Filter Stack Performance

FRD uses the first stack (i.e., filter stack) to maintain Class FS and IS blocks whose reuse distance is short. Figure 6 shows the hit ratio for the filter and reuse distance stacks. When a given cache size is small, the filter stack is more powerful. In prxy_0 workload with an 8-MB cache size, approximately 35% of the cache hit occurred in the filter stack, which is only 10% of the whole cache size. For the 1- and 2-GB cases, a cache hit occurred only from the reuse distance stack. In this case, the size of the reuse distance stack was sufficiently large to contain all blocks in the workload. In hm_0 workload, approximately 20% of the cache hit occurred in the filter stack with only 10% of the whole cache size. This shows that the filter stack not only works as a filtering buffer but also works as a cache buffer.

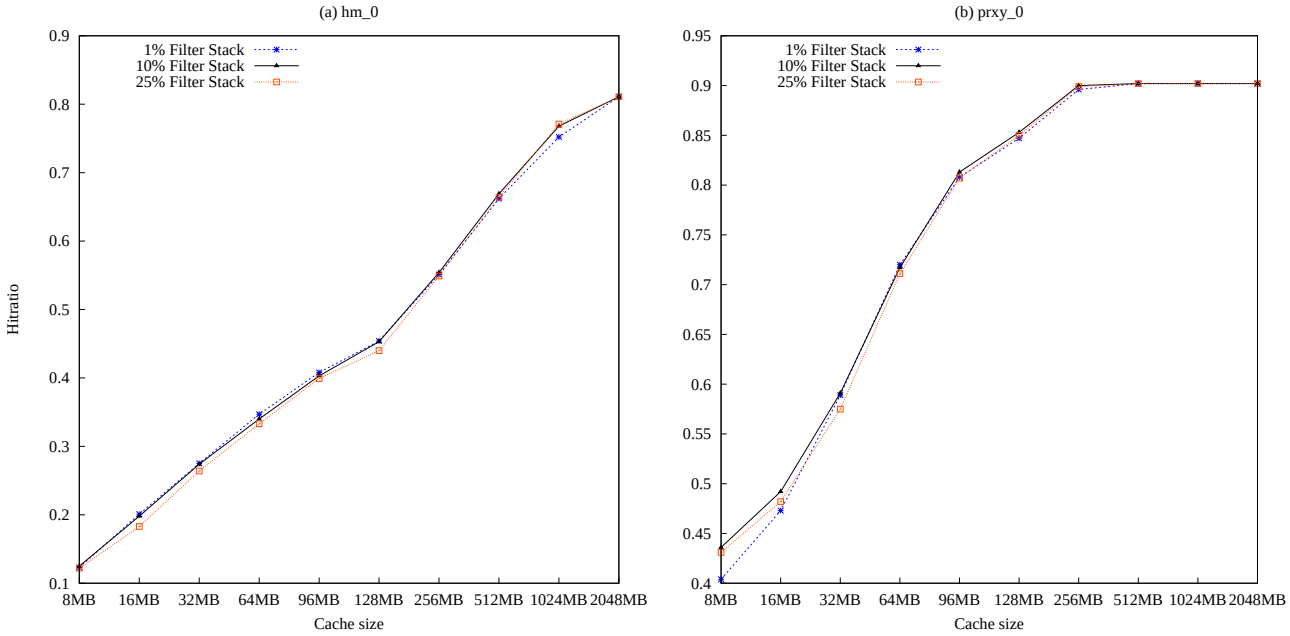

Figure 7. Sensitivity of the filter stack size from 1 to 25% of cache size. The variation in the hit ratio curve is not sensitive to the filter stack size.

E. Sensitivity and Overhead Analysis

The filter stack is a tunable parameter for the FRD cache algorithm. If we set the value to 100% of the cache, the FRD will yield the same result as in the LRU cache because the filter cache itself does not maintain any history blocks. In addition, if the size of the filter stack is too small, the hit ratio will be slightly lower because the blocks having a short reuse distance (i.e., Classes FS and IS) will be missed at the first reuse request. However, this does not generate a considerable performance gap because subsequent requests for the blocks will produce cache hits, as the blocks will be located in the reuse distance stack as a result of their history information. Figure 7 shows the sensitivity of the filter stack size from 1 to 25% of the cache size. As shown, the size of the filter stack is not sensitive to the hit ratio. However, empirically, 10% show the best performance among the many cache sizes. In the reuse distance stack, history blocks are maintained. Although the number of history blocks of the FRD can be greater than those in the ARC's ghost buffers, which is twice the size of a given cache, it is not greater than LIRS's LIR stack size because LIR uses 99% of a given cache size and we use 90% of given cache size. In addition, the history block is much smaller than the resident block because a history block contains only a block number. For a system with a 4-KB block size and 64-bit addressing, a history block (8 bytes) resides in only 0.2% of a resident block (4 KB).

VI. Discussion and Future Work

The workload proj_3 is an LIRS-friendly workload that includes many blocks with a long reuse distance. In an LIRS algorithm, when a new resident block is inserted into the LIR stack, the oldest resident block in the LIR stack will be moved to the HIR stack. This operation lengthens the life of a block and this is suitable for a workload with a long reuse distance. However, in the FRD cache, because the two LRU stacks have their own eviction points, such an operation is not possible. Instead, if we decrease the size of the filter stack, we can achieve a similar effect whereby a block's life is lengthened.

By contrast, OLTP, Web12, and Web07 are not LIRS-friendly workloads because they contain many blocks with infrequently accessed short reuse distances. LIRS does not filter out these blocks. As shown in Figure 5, ARC or FRD can effectively handle this kind of workload because ARC will increase its T1 buffer to maintain blocks with a short reuse distance, and FRD will effectively exclude such blocks from being inserted into the reuse distance stack.

The experimental results of ARC show a low hit ratio for the workloads whose reuse distance is greater than a given cache size such as proj_0, proj_3, or wdev_0. This limitation derives from the limited history buffer. Although ARC has this limitation, its adaptive-resizing mechanism effectively balances the variety of workload characteristics. An adaptive-resizable filter stack represents our continuing work. If we can apply this feature to FRD, it will achieve a higher hit ratio for an increasing variety of workloads.

VII. Related Work

Because a buffer cache algorithm is a traditional topic of research, many studies have been conducted. Traditionally, the LRU and Clock [14] algorithms, the latter being an approximation of LRU, are the most widely used. LIRS and ARC also have Clock-based designs known as Clock-Pro [15] and CAR [16], respectively.

To solve the problem of LRU cache, many previous studies offered history-based approaches. Many cache algorithms such as LRU-2 [4], 2Q [5], LRFU [9], EELRU [8], LIRS [1], and

ARC [2] use history information for evicted blocks. In most cases, this type of approach outperforms the LRU algorithm. However, this kind of algorithm concentrates on blocks that should be cached in the buffer. As seen, infrequently accessed blocks can be cached based on their policies and this generates cache pollution.

Frequency based approaches such as LFU or FBR [10] also suffers from cache pollution as a result of blocks having a high reference count but no recent access.

Other approaches exist that detect workload patterns on various I/O path levels. SEQ [17] detects patterns at a block level, whereas UBM [7], AFC [18], and DEAR [19] detect patterns on both file and application levels. In addition, program-counter or context-based approaches such as PCC [20] and AMP [21] have been proposed. They detect and classify sequential, looping, and other references simultaneously. However, this kind of algorithm has many tunable parameters or functions to detect various workloads. In addition, this kind of approach requires expensive computational resource to detect pre-defined patterns.

Recent cache algorithms such as CIO-LRU or CIO-ARC [22] support collective IO for HPC environment. However, these algorithms also require an additional external pattern detection module which introduces additional overhead. Multi-level buffer cache algorithm such as MQ [6], RED [23], PROMOTE [24] are introduced for multi-tiered storage. These algorithms concentrate on cache exclusiveness between multi-leveled cache buffers.

VIII. CONCLUSION

In this study, we proposed a buffer cache algorithm called FRD that considers both frequency and reuse distance. The primary purpose of the proposed algorithm is to exclude infrequently accessed blocks that may cause cache pollution and to maintain frequently accessed blocks based on reuse distance. Experimental results from our study showed that the proposed algorithm outperformed state-of-the-art cache algorithms such as ARC or LIRS and that FRD's hit ratio was stable for various cache sizes. The two-LRU stacks-based design enables easy implementation and low-eviction overhead with O(1) complexity.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MEST) (No. 2011-0016972) and supported by Brain Korea 21 PLUS project for POSTECH Computer Science & Engineering Institute.

REFERENCES

[1] Jiang, Song, and Xiaodong Zhang. "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance." ACM SIGMETRICS Performance Evaluation Review 30.1 (2002): 31-42.

[2] Megiddo, Nimrod, and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." FAST. Vol. 3. 2003.

[3] Jiang, Song, et al. "DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality." Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies. Vol. 4. 2005.

[4] O'neil, E.J., O'neil, P.E., and Weikum, G.: 'The LRU-K page replacement algorithm for database disk buffering', ACM SIGMOD Record, 1993, 22, (2), pp. 297-306

[5] Johnson, T., and Shasha, D.: '2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm', Proceedings of the 20th VLDB Conference. 1994.

[6] Zhou, Yuanyuan, James Philbin, and Kai Li. "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches." USENIX Annual Technical Conference, General Track. 2001.

[7] Kim, Jong Min, et al. "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references." Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4. USENIX Association, 2000.

[8] Smaragdakis, Y., Kaplan, S., and Wilson, P.: 'The EELRU adaptive replacement algorithm', Performance Evaluation, 2003, 53, (2), pp. 93-123

[9] Lee, D., Choi, J., Kim, J.-H., Noh, S.H., Min, S.L., Cho, Y., and Kim, C.S.: 'LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies', IEEE transactions on Computers, 2001, (12), pp. 1352-1361

[10] Robinson, J.T., and Devarakonda, M.V.: 'Data cache management using frequency-based replacement' (ACM, 1990. 1990)

[11] cache2k - High Performance Java Caching, Benchmarks for cache2k, https://github.com/cache2k/cache2k-benchmark

[12] Narayanan, D., Donnelly, A., and Rowstron, A.: 'Write off-loading: Practical power management for enterprise storage', ACM Transactions on Storage (TOS), 2008, 4, (3), pp. 10

[13] Belady, L.A.: 'A study of replacement algorithms for a virtual-storage computer', IBM Systems journal, 1966, 5, (2), pp. 78-101

[14] Corbato, F.J.: 'A paging experiment with the multics system', In Honor of P. M. Morse, 1969, MIT Press, pp. 217–228

[15] Jiang, Song, Feng Chen, and Xiaodong Zhang. "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement." USENIX Annual Technical Conference, General Track. 2005.

[16] Bansal, Sorav, and Dharmendra S. Modha. "CAR: Clock with Adaptive Replacement." FAST. Vol. 4. 2004.

[17] Glass, Gideon, and Pei Cao. Adaptive page replacement based on memory reference behavior. Vol. 25. No. 1. ACM, 1997.

[18] Choi, Jongmoo, et al. "Towards application/file-level characterization of block references: a case for fine-grained buffer management." ACM SIGMETRICS Performance Evaluation Review. Vol. 28. No. 1. ACM, 2000.

[19] Choi, Jongmoo, et al. "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme." USENIX Annual Technical Conference, General Track. 1999.

[20] Chris Gniady , Ali R. Butt , Y. Charlie Hu, Program-counter-based pattern classification in buffer caching, Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, p.27-27, 2004.

[21] Z Zhou, Feng, J. Robert von Behren, and Eric A. Brewer. "AMP: Program Context Specific Buffer Caching." USENIX Annual Technical Conference, General Track. 2005.

[22] Lu, Yin, et al. Revealing applications' access pattern in collective i/o for cache management. In: Proceedings of the 28th ACM international conference on Supercomputing. ACM, 2014. pp. 181-190.

[23] Zhao, Yingjie; Xiao Nong; Liu, Fang. Red: An efficient replacement algorithm based on REsident distance for exclusive storage caches. In: Mass Storage Systems and Technologies (MSST), 2010

[24] B. S. Gill, "On Multi-level Exclusive Caching: Offline Optimality and Why promotions are better than demotions", in Proc. FAST Conf. , 2008. pp. 49-65.