2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

KBP: Mining Block Access Pattern for I/O Prediction with K-Truss

Jia Ma, Xianqi Zheng, Yubo Liu, Zhiguang Chen*
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, Guangdong 510006, China
Email: {majia5, zhengxq27}@mail2.sysu.edu.cn, {yubo.liu, zhiguang.chen}@nscc-gz.cn

Abstract—**Block prefetching is a technology widely used to improve the I/O efficiency of storage systems. Block access pattern prediction is a key part of the prefetching algorithm. However, existing block access pattern prediction methods cannot achieve the goals of low overhead, real-time, and self-adaptability at the same time. In this paper, we propose a real-time prediction method, called KBP (K-Truss-based Block access pattern Prediction). KBP uses SA (Sequential Access) Filter to identify and filter sequential access patterns to distinguish different patterns. Then, KBP uses K-Truss dense subgraph algorithm to detect compound access patterns, so as to use the time and space advantages of the K-Truss algorithm to reduce the overhead of access pattern recognition and make it possible for KBP to run in real time. Furthermore, KBP uses online reinforcement learning to achieve the goal of self-adaptability. We evaluate KBP in real-world workloads and the results show that KBP can improve the hit rate of 11.2% compared to the state-of-the-art prefetching algorithms on average.**

Index Terms—**prefetching, access pattern, K-Truss dense subgraph, block storage**

1. Introduction

Block storage is an important component of many systems. For example, many cloud computing systems use Ceph [1] to provide block storage service for visual machines. A popular technique to improve the performance of the block storage is to prefetch the strongly correlated data blocks into the fast cache layer. The main factor that affects the efficiency of prefetching is the ability to quickly and accurately mine block access patterns. The block access patterns contain two cases: simple and compound [2], [3]. In the simple case, blocks will be accessed in order or in stride, which is easy to be predicted and prefetched. However, there is no obvious rule for block access in the compound case. The compound case refers to an access sequence in which there is no regularity in the offset of blocks, but this access sequence often appears in the block access stream. The compound case brings a big challenge for the block access pattern prediction.

* *Corresponding author*

There are some studies research on the block access pattern perdition in the compound case. However, they are sub-optimal: some solutions (e.g., [4], [5], [6]) use a single perdition mechanism in all cases, which makes them difficult to work accurately and effectively in a variety of workloads; although some solutions (e.g, [3]) consider the variety of workloads, they take high latency and space overhead; some solutions (e.g., [7], [8]) improve the efficiency of prediction by allowing the application to provide additional hints, but this requires modifying the application.

In order to solve the shortcomings of existing block access pattern prediction algorithms, we follow three design goals: **1) Versatility.** The prediction algorithm should work well in both simple and compound cases. **2) Low overhead and real-time.** The prediction process cannot bring too much overhead so that it can run in real time. **3) Adaptability.** Due to the access pattern in the compound case may change as the application is running, this requires that the prediction algorithm needs to be adaptive.

In this paper, we propose KBP, a block access pattern prediction algorithm to meet the design principles above. To achieve the versatility goal, KBP uses SA (Sequential Access) Filter to divide the data access flow into simple and compound cases, and uses different prediction algorithms for different cases. This design makes KBP applicable to any workload.

To achieve the low overhead and real-time goal, the big challenge of KBP is to efficiently predict the access pattern in the compound case. Although the block access pattern in the compound case is not continuous, the access distance between blocks still follows certain rules, which we call the semantic. When KBP detects an access sequence is the compound case, it uses a graph to abstract these semantics and uses edges to represent the relationship between the blocks. Then, KBP decomposes the semantic graph into multiple communities by K-Truss algorithm [9], and each community represents a recognized compound pattern. Benefit from the low overhead of relevant algorithms (e.g., [10], [11], [12]), KBP has low prediction overhead in the compound case and makes it possible to meet the real-time requirements.

To achieve the adaptability goal, KBP uses online reinforcement learning [13] to adapt to the workload change. When KBP detects that the hit rate drops, it means that the current access pattern may not fit the workload change and KBP will reduce the number of the prefetch blocks. By reducing the number of prefetch blocks, KBP can adapt to the new access pattern again. Online reinforcement learning is used to predict the relationship between the number of prefetch blocks and the hit rate, which makes this adaptive process more efficient. Also, if the hit rate drops for a while, it indicates that most of the access patterns identified by KBP are already invalid. Therefore, the KBP decay strategy is used to create a new KBP to replace the old one and relearn the new access pattern.

Our paper makes three contributions.

1. KBP combines SA Filter and K-Truss to allow the prediction algorithm to be adapted to both simple and compound cases, while only taking low overhead.

2. KBP adopts online reinforcement learning and KBP decay strategies to make it can quickly adapt to various workloads.

3. We implement KBP and evaluate it on the real traces. The experimental results show that KBP boosts the cache hit ratio by up to 2 times over state-of-the-art cache strategies ARC [14] and improves over the sequential prefetching algorithm AMP by 11.2% on average.

The rest of the paper is organized as follows. Section 2 presents the background and our motivation for proposing KBP. Section 3 describes the design of KBP and the cache prefetching and replacement strategies. Section 4 reports the experimental results. Section 5 discusses related works and the last section concludes the paper.

2. Background and Motivation

2.1. Block Prefetching Algorithm

Prefetching, as a technique with significant effects to improve storage system cache performance, has been widely used in a large number of systems.

The most common prefetching approach is to perform sequential readahead (e.g, [15], [16], [17]). For example, the well-known prefetching algorithm [16] in the Linux Kernel can efficiently deal with the pattern from a single sequential stream and adaptive algorithms such as AMP [18] recognize multiple sequential patterns at the same time. Although these algorithms are widely used, their effect is limited.

Stride-based prefetching (e.g, [7], [8]) has also been studied where strides are detected based on information provided by the application or a lookahead into the instruction stream. However, most strides lie within the block size so it is usually not better than sequential prefetching.

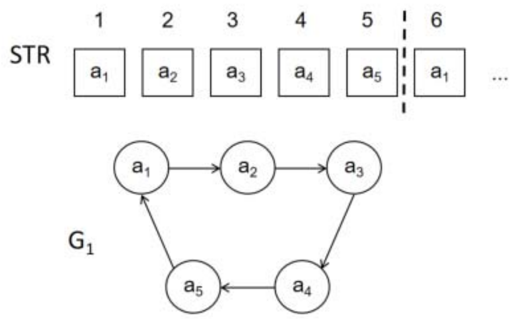
Due to various data structures such as trees, matrices, and user or program access behavior, there are rules of block access patterns. Therefore, pattern-aware prefetching algorithms are proposed in various forms. For example, C-Miner [3] uses frequent subsequences to mine access patterns offline, but this method is not real-time and adaptive. Another type is to establish the relationship between blocks with probability graph (e.g, [19], [20]). However, the space overhead increases as the number of blocks increases.

Therefore, it is important to propose an online low-overhead adaptive prefetching algorithm that can recognize both simple and complex patterns.

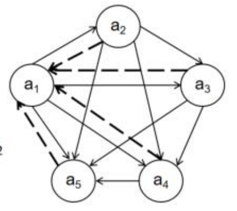
2.2. K-Truss Discovery

A graph is a common data structure that consists of a finite set of vertices and a set of edges connecting them. Since many real-life problems involve the representation of the problem space as a network, graphs have become increasingly important. Especially with the emergence of big data, graph algorithms are gradually being applied to the storage system. For example, DAG refers to "Directed Acyclic Graph" which is used in Hadoop and Spark to improve performance.

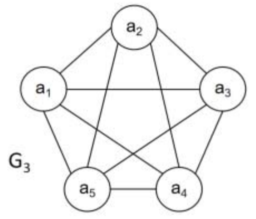
Among them, the dense subgraph algorithm has attracted much attention. Among these algorithms, a popular dense subgraph concept that has recently been studied is K-Truss. Because K-Truss achieves a balance between the structure cohesiveness and computational efficiency and it can handle dynamic graphs easily [21]. In this paper, we use K-Truss to analyze offline traces and find the algorithm is useful for mining the block access patterns.



(a) Sequential Stream STR and Single Ordered Ring



(b) Directed Graph and History Window is 4



(c) Undirected Graph of (b)

Figure 1. Sequential Access And Semantic Graph

2.2.1. What's K-Truss. We use $G = (V_G, E_G)$ for an undirected, simple unweighted graph where V_G stand for the vertex set and $E_G \subseteq V_G \times V_G$ are edge set. And the support of an edge is the number of triangles formed by this edge in the graph. A K-Truss is the largest subgraph of the graph that each edge is contained in at least k-2 triangles within this subgraph [9].

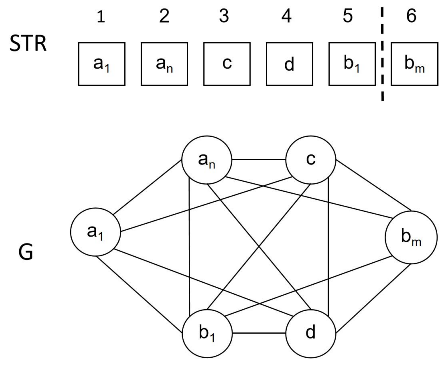


Figure 2. Stream STR after sequential filtering And Semantic Graph

The larger k is, the more cohesive the graph is. For the graph G in Figure 1[c], we can easily figure out it is a 5-truss because the minimum support is 3. For the graph G in Figure 2, although $e(a_n, c)$ is contained in $\triangle_{a_n c b_1}$, $\triangle_{a_n c d}$, $\triangle_{a_n c a_1}$ and $\triangle_{a_n c b_m}$ which means the support of it is 4, the minimum support in G is 3. Thus G is still 5-truss.

2.2.2. K-Truss discovery on offline traces. We use K-Truss to analyze the actual workload trace and find that the dense subgraph of K-Truss also exists in the network structure formed by the blocks. Considering the sequential access sequence STR in Figure 1 (a) and assuming that each adjacent I/O access request is associated, we can use a directed edge to show that they are related. In this way, we can get the G_1 which is in the shape of a single-stranded ordered ring in Figure 1(a).

However, in the real workload trace, all the I/O requests from one or more processes will constitute a long I/O access stream, hence there will be other I/O requests in the middle of the sequential access sequence. As a result, we can not recognize it as sequential access easily. Therefore, researchers often use history window to record n history requests and the current request block is considered to be related to the block in history window. When the size of the history window is 4 and a_1 is requested for the second time in Figure 1(a), then a_1 is related to a_2, a_3, a_4 and a_5 which is shown in dotted lines in graph G_2 in Figure 1(b) Converting G_2 in Figure 1(b) to the undirected graph G_3 in Figure 1(c), we can find G_3 is a 5-ktruss and G_3 better reflects the correlation between these blocks than G_1.

K-Truss can recognize the switching between sequential sequences. Through sequence recognition and filters, we only put the first and the last blocks of the sequential access sequence and nonsequential blocks in the history window. As shown in Figure 2, blocks a_1, a_n in STR are the first and the last blocks of the sequential access sequence like $a_1, a_2, \cdots, a_n$. So are b_1 and b_m. Then we can get the correlation graph G which shows if a_n is accessed recently, b_1 is likely to be accessed in a short period and this helps to connect two sequential sequences and reduce cache missing rate.

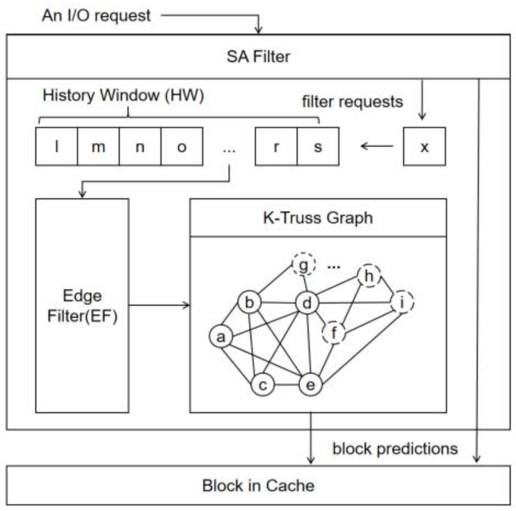


Figure 3. Architecture of KBP

Take the Online trace as an example in Figure 8, which consists of several sequential access sequences that are constantly switching. Although AMP is very good at this kind of trace data and it can effectively increase the hit rate, AMP suffers from many caches missed during the switch of two sequential sequences. This problem can be avoided because the switching of Online is fixed. And KBP can discover these access patterns so KBP increases the hit ratio of AMP by 71% when the cache size is 256M.

3. The KBP Algorithm

KBP mines block access patterns on the fly along with the I/O access stream going and makes efficient I/O predictions. There is no need to modify the lower-level applications or I/O libraries. As shown in Figure 3 it is the structure of KBP. Specifically, when access comes, SA Filter is used to confirm whether current access sequence is sequential or not. If not, the access will enter the compound pattern mining part of KBP and establish an edge with blocks maintained in the HW(History Window). Then EF(Edge Filter) is used to record the frequency of each edge and only edges with frequencies exceeding the Edge_threshold are added to the K-Truss correlation graph. Then we use K-Truss decomposition algorithm to decompose the graph into dense subgraphs in order to discover compound access patterns. Thus, KBP can provide one or more I/O predictions results if current request satisfies the identified pattern. In addition, KBP updates the whole architecture through the KBP decay algorithm to ensure that it can adapt to new workloads in real-time and provide more accurate I/O predictions.

3.1. Block Access Pattern Mining

We introduce earlier that the access pattern is divided into sequential and compound patterns in our paper. Since there are many excellent algorithms for identifying sequential patterns such as AMP, and the sequential pattern can bring some recognition burden and interference to the detection of compound patterns, so we use a SA Filter in KBP to identify and filter sequential sequences for I/O requests. And K-Truss is mainly used to recognize compound patterns.

SA Filter inherits the core logic of AMP, which considers every single access as a potential sequential access sequence to discover multiple sequential access patterns synchronously [2]. Because SA Filter maintains the meta of recent potential sequential access blocks, it can detect a sequential access sequence even it is broken by access from other processes, and adapt to the dynamic expansion of each sequential access sequence. More details can be found in [18]. The access stream described later in this section refers to the stream after SA Filter.

Many researchers have proposed that there may be associations between blocks that are close to each other in the request stream. We define this association as a combination of $\{pre, next\}$, that is, a compound pattern of block access where pre and $next$ represent the address of blocks. This combination means when pre is accessed, $next$ is likely to be accessed soon.

We define the access distance [3] between two blocks as the unique I/O block requests which are different from these two blocks both. In this way, a compound pattern is measured according to the principle that two blocks are likely related only if the access distance between them is not greater than a predefined value. We use CorrDis for this predefined value. Therefore, the first condition for two blocks to form a compound pattern is that the access distance between them is less than CorrDis.

In the implementation of KBP, it maintains a global index to label every block in the storage system. We use HW(History Window) to record the most recently accessed blocks in the history I/O stream. HW is a queue with the fixed size of CorrDis and is managed in the FIFO(First In First Out) manner to preserve the access order of blocks. HW is used to buffer the previous CorrDis accessed blocks of the current requested block. When a new block request b_i arrives, it will build compound pattern $\{hw_i, b_i\}$ with the block $\{hw_1, \cdots, hw_{CorrDis}\}$ in HW. Consequently, any two blocks in HW can constitute a compound pattern.

All these combinations produced by HW are regarded as a directed edge recorded in EF(Edge Filter) in the format of $\{src, des\}$. And it is obvious that the number of edges in EF is large and a lot of these combinations are meaningless. So each edge in EF corresponds has a counter Count to record the frequency of the edge. When this pattern appears again, Count will be increased by 1. At the same time, we introduce a new variable EdgeThreshold to filter edges, and only those edges whose Count is larger than EdgeThreshold can be added to the K-Truss graph. EdgeThreshold is to ensure that the detected compound pattern had a certain persistence.

The value of EdgeThreshold is related to the frequency of blocks in the stream. In the cache replacement policy LFU, blocks with high-frequency will be saved in the MFU position of the cache which is similar to other strategies such as LRU and ARC. Therefore, we don't need to consider the impact of these high-frequency blocks. Also, most blocks are accessed only once and these blocks are meaningless for us to explore the compound. So in subsequent experiments, the value of EdgeThreshold is the average of the block frequency.

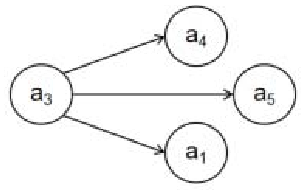


Figure 4. Directed spanning tree with a_3 as root in Figure 1

K-Truss graph is regarded as an undirected graph and it can be decomposed into truss communities using the K-Truss decomposition algorithm [12]. A truss community represents a cluster of associated blocks. In addition, through the K-Truss dynamic update algorithm [10], [11], the communities discovered by the K-Truss graph can be updated in real time. The process of using K-Truss graph to query a block b as pre in compound patterns is as follows. First, call $KTrussQuery(b)$ in [10], [11], [12] with b and obtain the largest K-Truss community related to b. Secondly, through the directed edges records in the EF, a DST(directed spanning tree) with b as root can be generated from the truss community. For example. Figure 4 is the DST of a_3 in the largest 5-truss Figure 1. Finally, the DST can provide I/O prediction when the specified block is accessed.

In summary, there are two steps to detect the access pattern in KBP. The first is the recognition of the sequential pattern. SA Filter base on AMP can recognize sequential sequences and filter I/O stream. The second is the recognition of compound patterns. The current requested block b is correlated with blocks in HW with a fixed-length CorrDis to generate an edge and then we increase the Count of each edge by 1 in EF. When the Count reaches EdgeThreshold, the edge will be added to the K-Truss graph and the K-Truss communities will be updated with a dynamic update algorithm. Taking the K-Truss query algorithm and the help of records in EF, we can get the I/O predictions of b.

3.2. Storage System Optimization with KBP

From Section 3.1, we can explore data access patterns and provide I/O predictions. Then, we propose data prefetching and replacement strategies to optimize storage system performance.

3.2.1. Prefetching. An important application for exploring block access patterns is to improve the performance and HR(hit ratio) of cache [18]. Common prefetching strategies include passive prefetching and active prefetching. Passive prefetching means prefetch the subsequent blocks that constitute a known access pattern to cache when an I/O request

Algorithm 1: Prefetching Algorithm of KBP

```
Data: Cache $C$; Trigger max $trigger\_max$;
  Active prefetching trigger $Trigger$; Current time $t$;
  Learning rate at time(t) $\lambda_t$;
  Average hit-rate at time(t) $HR_t$;
  Average prefetched data hit-rate at time(t) $PHR_t$;
  Learing rate update interval $i$;
Input: Request block $q$
1  if $q \in C$ then
2      $C$.UpdateDataStructures($q$);
3      if $q$ is prefetched from KBP then
4          UpdateTrigger($Trigger$, $\lambda$, $trigger\_max$, 1);
5      end
6      if $t$ % Trigger == 0 then
7          KBPPrefetch(KTrussQuery($q$));
8      end
9  else
10     if $q \in KTruss$ then
11         UpdateTrigger($Trigger$, $\lambda$, $trigger\_max$, 2);
12     end
13     C.ADD($q$);
14     KBPPrefetch(KTrussQuery($q$));
15     UpdateKTrussStructure($q$);
16 end
17 BlockFreq($q$) = 1 + BlockFreq($q$);
18 if $t$ % $i$ == 0 then
19     UpdateLearningRate($PHR_t$, $PHR_{t-i}$, $HR_t$, $HR_{t-i}$, $\lambda_{t-i}$, $\lambda_{t-2i}$);
20 end
```

Algorithm 2: UpdateTrigger($Trigger$, λ, $trigger\_max$, *updateway*)

```
1  $\triangle_{Trigger}$ = $trigger\_max$ - $Trigger$;
2  if updateway == 1 then
3      $Trigger$ = max($Trigger$ $\times e^{-\lambda}$, 1);
4  else
5      $\triangle_{Trigger}$ = max($\triangle_{Trigger} \times e^{-\lambda}$, 1);
6  end
7  $Trigger$ = ceil($trigger\_max \times \frac{Trigger}{Trigger+\triangle_{Trigger}}$;
```

is missing. Active prefetching is triggered when the current request block is found to be consistent with the discovered compound pattern information. In KBP, passive prefetching is the same as in AMP. And active prefetching is provided according to the truss structure of the compound pattern recognition based on the K-Truss graph. When a part of blocks in the same truss is hit, it is reasonable to prefetch the rest of this truss into the cache.

However, the K-Truss graph is constantly changing with the I/O request flow. Keeping a snapshot will take up a lot of space overhead, and calculating the hit ratio of a truss will also bring more time overhead. Therefore, we introduce a new variable $Trigger$ to control the interval for active prefetching. Its meaning is the best time for active prefetching and is adjusted by online learning with regret minimization [22], [23].

The proposed cache prefetching strategy using KBP for I/O predictions is shown in Algorithm 1 and only compound patterns need to be considered. When requested block q is hit in C(Lines 1-8), the structure of cache C will update according to the cache replacement strategy adopted by C(Line 2). If q is previously prefetched from the K-Truss graph, this shows that the prefetching strategy is meaningful. Therefore, the UpdateTrigger is used to reduce $Trigger$ to increase the frequency of active prefetching(Lines 3-5). When active prefetching is triggered, use KTrussQuery(q) to query and get the max K-Truss of q. And then use KtrussPrefetch to prefetch DST rooted at q into the cache(Lines 6-8). When q is missing in C(Lines 9-20), first determine whether q is included in the K-Truss graph. If q is included in the K-Truss graph and not prefetched, it indicates the current prefetching strategy is not effective, so we need to increase $Trigger$ appropriately through UpdateTrigger to reduce the number of prefetches(Lines 10-12). Then fetch block q into the cache(Line 13) and repeat the previous operations(Line 14). At the same time, add q into the recognition process of the compound access pattern recognizer to update the KBP structure(Line 15). Also, the frequency of every block is recorded in BlockFreq(Line 17) which provides the selection of the EdgeThreshold value in the subsequent KBP attenuation. The learning rate λ related to the adjustment of $Trigger$ will update every update interval(Lines 18-20). The algorithm UpdateLearningRate is explained in Algorithm 3.

The adjustment method of $Trigger$ is shown in Algorithm 2, which is an idea of reinforcement learning by adjusting $Trigger$ according to the prefetch effect. First, calculate the difference between $trigger\_max$ and $Trigger$(Line 1). When the update way is 1, decrease the value of $Trigger$(Lines 2-3). Otherwise, increase $Trigger$ size by reducing the difference(Lines 4-6). Finally, normalize $Trigger$(Line 7).

3.2.2. Replacement. Another application for I/O prediction is the strategy of cache replacement. It is mainly reflected in the 7 and 14 lines in Algorithm 1. In line 7, if the prefetch block b already exists in C, the importance of b in the structure of cache will be increased through the caching replacement policy. This helps to sufficiently handle churn workload which is repeated access to a subset of stored items in which each item is accessed with equal probability. In line 14, we replace the older truss that already exists in the cache with prefetched truss as much as possible.

3.3. Dynamic and Adaptive Decay of KBP

On the one hand, due to the continuous I/O requests stream and the fixed value EdgeThreshold in EF, this will cause the number of edges recorded in EF to expand and take up a lot of memory. Also, the count of each edge will

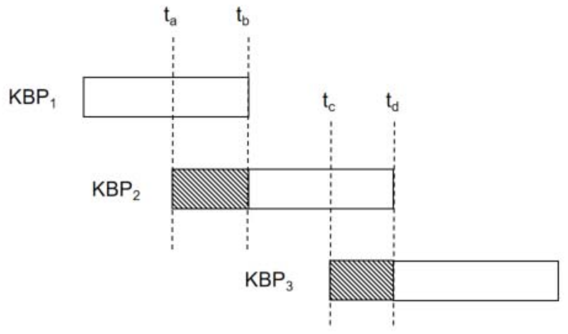


Figure 5. Decay of KBP

eventually exceed EdgeThreshold and they will be added to the K-Truss graph which will cause the graph to eventually form a complete graph and become meaningless.

On the other hand, due to changes in user behavior and block modifications, the correlation among blocks has changed and this makes KBP contain many meaningless compound patterns. This will reduce the accuracy of I/O predictions. Therefore, we need to create a new KBP at the appropriate time to rediscover new access patterns.

If the old KBP is directly replaced by the new KBP, the new KBP is unable to provide I/O predictions in the first period because the new K-Truss graph is empty. So the new KBP needs some time to master new access patterns. Therefore, K-Truss attenuation is proposed to clean up redundant invalid patterns and help KBP to provide more accurate I/O predictions. As shown in Figure 5, t_a represents the moment when KBP_1's I/O prediction effect is not good, so a new KBP_2 is created to explore new access pattern. t_b indicates that the effect of KBP_1 is very poor and KBP_2 has enough information to provide the I/O predictions. At this time, KBP_2 is used to replace KBP_1. Note that during the period from t_a to t_b only KBP_1 provides I/O predictions. It's same during the period from t_c to t_d.

The prediction effect of KBP is reflected in its I/O prediction accuracy. PHR(Hit Ratio of Prefetched data) is important to HR. For example, when PHR is less than HR, it means that current prefetch rules are meaningless and it lowers HR. Therefore, The changes of PHR and HR are used to assist KBP attenuation. The adjustment of the learning rate λ is also included in this process.

Algorithm 3 is the learning rate update and K-Truss attenuation method. The changes of PHR, HR and learning rate over the previous two intervals are calculated respectively(Lines 1-3). When the change of learning rate is not zero, adjust the λ through gradient descent(Lines 4-7). First, the gradient of the performance(average hit-rate) concerning the learning rate over the previous two windows is calculated. If the gradient is positive(negative, resp.), then the direction of the change of the learning rate is sustained(reverse, resp.)(Line 5). If the performance increase(decrease, resp.) the learning rate by an amount proportional to the learning rate change relative to the previous window(Line 6). The learning rate is initialized randomly between 0.1 and 1.

Algorithm 3: UpdateLearningRate(PHR_t, PHR_{t-i}, HR_t, HR_{t-i}, λ_{t-i}, λ_{t-2i})

```
1  $\triangle_{PHR}$ = $PHR_t$ - $PHR_{t-i}$;
2  $\triangle_{HR}$ = $HR_t$ - $HR_{t-i}$;
3  $\triangle_{\lambda}$ = $\lambda_{t-i}$ - $\lambda_{t-2i}$;
4  if $\triangle_{\lambda} \neq 0$ then
5      sign = +1 if $\frac{\triangle_{HR}}{\triangle_{\lambda}} > 0$ else -1;
6      $\lambda_t$ = max($\lambda_{t-i} + sign \times |\lambda_{t-i} \times \triangle_{HR}|$, 0.1);
7      unlearnCount = 0;
8  else
9      if unlearnCount < 10 and ($\triangle_{PHR} > 0$ or $\triangle_{HR} > 0$) then
10         unlearnCount = 0;
11     else
12         if $\triangle_{PHR} < 0$ or $PHR_t < HR_t \times \alpha$ then
13             unlearnCount = unlearnCount + 1;
14         end
15     end
16     if unlearnCount == 10 then
17         StartNewKBP(BlockFreq);
18     end
19     if unlearnCount >30 then
20         SwitchKBP();
21         unlearnCount = 0;
22         $\lambda_t$ = choose randomly between 0.1 & 1;
23     end
24 end
```

We use unlearnCount to record the number of times the learning rate has not changed. When unlearnCount is less than 10 and the $\triangle PHR$ or $\triangle HR$ is greater than 0, it shows that although the learning rate has no change for some time, the scheme is still effective(Lines 9-10). Otherwise, when the PHR decreases or the PHR is less than the HR by α, unlearnCount will increase automatically(Lines 11-15). Then if unlearnCount reaches 10, we use StartNewKBP(BlockFreq) to start a new KBP and the value of EdgeThreshold in the new KBP is confirmed according to the frequency of blocks currently recorded(Lines 16-18). When unlearnCount is larger than 30, it calls SwitchKBP to replace the old KBP with the new one to complete the attenuation. At the same time, reset unlearnCount and λ(Lines 19-23).

In summary, the learning rate is adjusted by the gradient descent method [24] to effectively adapt to the various workload. And the KBP attenuation can maintain the efficiency of I/O prediction.

4. Evaluation

4.1. Experimental Setup

To measure the I/O prediction ability of KBP, we conducted a series of comparative experiments which are all

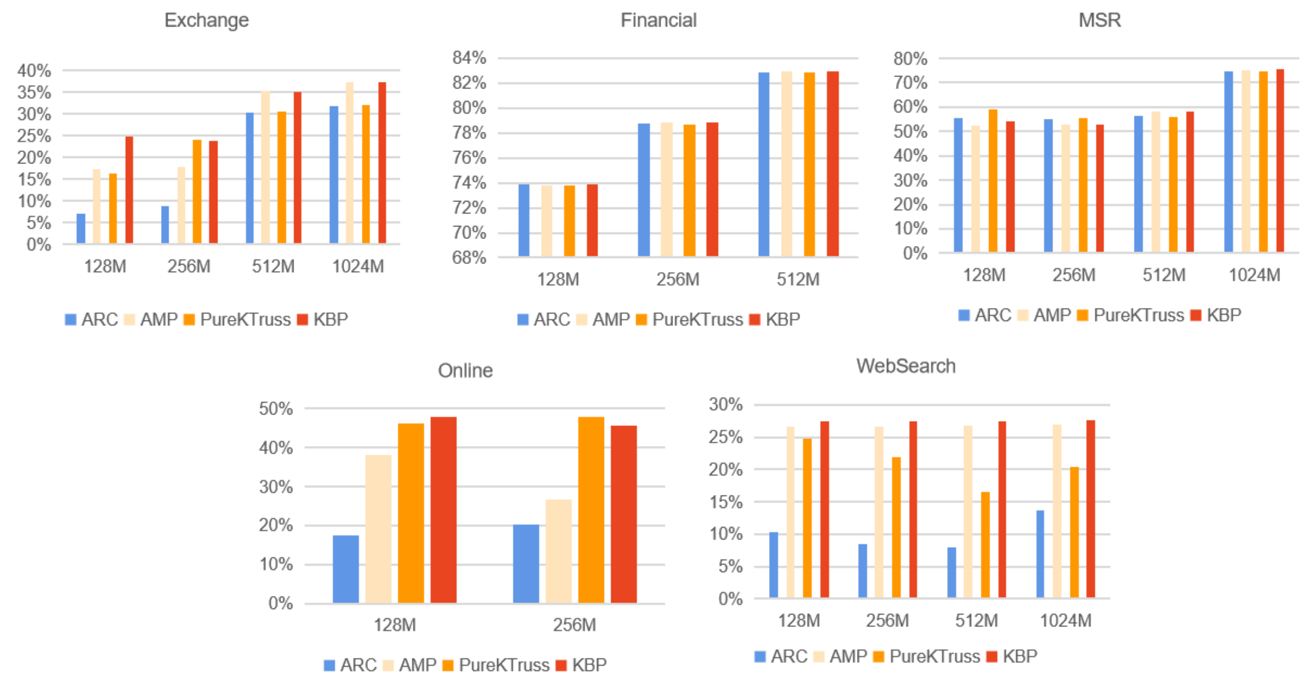


Figure 6. Hit Ratio of 4 algorithms on 5 traces

based on the ARC caching replacement scheme. In our experiments, we assume that the requested blocks, whether predicted or on-demand, are directly inserted into the cache without considering lower-level storage devices.

Algorithms. We compared KBP against 3 algorithms: ARC [14], AMP [18], and PureKTruss. In many papers (e.g., [2], [18]), AMP is better than other algorithms such as MMS [20] and C-Miner [3] in most cases. Therefore, we choose AMP for comparative experiments. In addition, we use PureKTruss which is KBP without SA Filter to compare with AMP [18] and KBP. PureKTruss treats all patterns as compound patterns.

Workloads and Simulations. We use 5 common and widely used data sets for experiments. Experiments are carried out under various cache sizes to verify that KBP can adapt to various cache sizes and workloads. The details of each trace are shown in table 1. We can find the average frequency is big enough to effectively recognize patterns in the I/O stream. The initial value of $Trigger$ is 1 by default.

Experimental Indicators. The most important indicator is cache hit ratio. Also, there are two criteria in the cache prefetching strategy [3].

1. Avoid prefetching waste. Do not prefetch blocks that will be removed from the cache before being used.

2. Avoid cache pollution. Do not cause other unused blocks to be removed due to prefetch blocks, which will result in repeated I/O.

So we will also record prefetching used ratio, and increase-ratio of the I/O and cache HR.

TABLE 1. DETAILS OF DIFFERENT WORKLOAD TRACES

Traces	Total requests	Unique data size(MB)	Block Mid-frequency	Block Max-frequency
Exchange	2,995,366	9,083	3.68	17,269
Financial	5,585,220	1,078	12.8104	32,135
MSR	8,586,983	2,262	17.21	152,770
Online	1,047,161	450	9.09	1,980
WebSearch	7,802,253	8,973	7.55	627

4.2. Performance Evaluation

As shown in Figure 6, it is the cache HR results of the four algorithms under different caches size on five traces. To summarize the findings, the smaller the cache size, the better the cache HR of KBP. Compared with ARC, the other three algorithms all have a significant improvement on Exchange, Online, and WebSearch traces. When the cache size is 128M, the average increase of AMP is 140.9%, while the average increase of PureKTruss is 135.5%. In contrast, KBP has a maximum average increase of 199.1%. Even at 512M, KBP has a maximum average increase of 65.4% which is still better than 63.4% of AMP. Comparing KBP with AMP, the average hit rate of KBP is 11.2% higher than that of AMP, especially when the cache is 256M on Online trace, which is the largest and 71% higher than that of AMP. In addition, the experimental results show that the effect of PureKTruss is not stable. Because the sequential sequences interfere with the block access patterns and increase the identification burden of PureKTruss. KBP combines the advantages of AMP and PureKTruss, so the hit rate is usually the best.

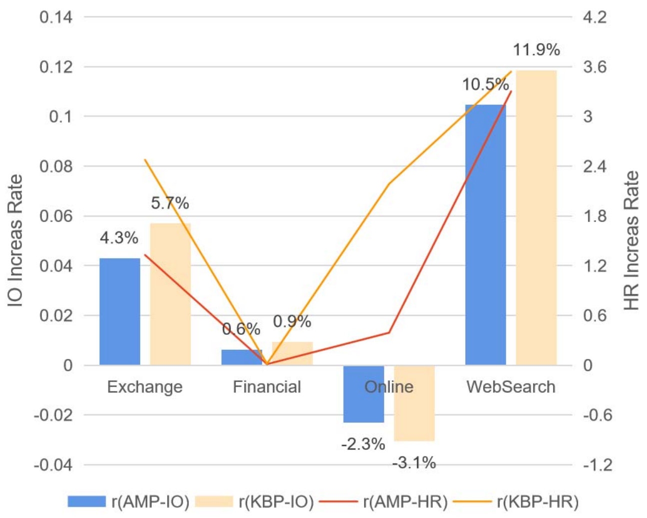


Figure 7. I/O and HR increase rate between AMP and KBP when cache size is 256M

We also considered the increased ratio of I/O and hit rate brought by I/O prediction and prefetching. Figure 7 represents the I/O increase ratio and HR increase ratio of AMP and KBP relative to the ARC on each trace when the cache size is 256M. The KBP's HR increase ratio curve is always above AMP and the I/O increase ratio curve of KBP on Exchange, Financial, and WebSearch are close to that of AMP. Especially on Online trace whose access pattern is shown in the top part of Figure 8, a normal algorithm will appear repeated I/O when the cache size is small because it is a churn workload. However, KBP can provide more effective prefetching and replacement strategies to reduce I/O times while significantly increasing HR than AMP.

In addition, we also record the utilization of the prefetch block. In most cases, the prefetched block utilization of AMP and KBP are similar. It is between 80%-90%, which is much higher than the total cache hit rate. The prefetched data utilization rate of KBP is significantly higher than that of AMP on Online. While in most cases, PureKTruss prefetching utilization is not good and PureKTruss performs more KBP decay.

4.3. Effect of SA Filter

In Figure 8, we discuss regions labeled A, B, C, and D in this figure. At A, PureKtruss has not learned enough information yet, so the overall hit rate is the same as ARC. The hit rate of AMP reaches the maximum during each sequential access, but the hit rate is minimized between two sequential access switches. KBP uses SA Filter to recognize sequential patterns and provide sequential I/O predictions, and with the help of K-Truss, the HR is still high during switching. At B, PureKTruss has learned enough information at this time, and the cache HR of PureKTruss starts to rise at this time. The effect of AMP is not good at

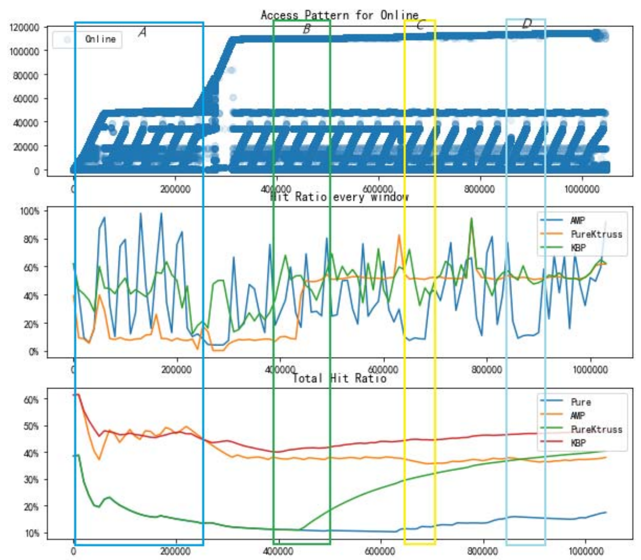


Figure 8. Effect of SA-Filter on Online and the cache size is 128M. The first plot is the block access diagram of Online, which is composed of many sequential access sequences. The second plot shows the cache HR of AMP, PureKTruss, and KBP in each window which size is 10,000 I/O requests. The last plot shows the changes of total HR of the four methods over the entire time period.

C and D due to much switching which results in that the overall HR of PureKTruss surpassed KBP. It can be found that in most cases, the overall hit rate of PureKtruss is lower than that of KBP. This is because SA Filter can effectively filter the effects of sequential mode and provide sequential I/O predictions so that KBP will not lose on the starting line. This proves that it is wise to distinguish between simple patterns and compound patterns and use different adaptive algorithms to mine them.

4.4. Effect of K-Truss and Active Prefetching

In Figure 9, we also discuss regions labeled A, B, C, and D in this figure. At A, the KBP cache HR in the second plot drops and is lower than AMP, resulting in a sudden increase of $Trigger$. By reducing active prefetching in this period, the total HR of KBP is still higher than AMP. At the same time, during this period, the HR of PureKTruss continues to increase rapidly, which shows that the K-Truss structure can contain both sequential and compound modes and K-Truss discovery is meaningful. At B, we can see that the trigger value is 1, and the cache HR of KBP and PureKTruss is higher than that of AMP during this period, which indicates that there are more compound sequences during this period, and the total HR of PureKTruss is even higher than that of AMP for a short time. At C, the overall HR of PureKTruss is decreasing, and there is more than 10 KBP decay in the process of C, which indicates that the pattern originally recognized has changed. In this period, KBP constantly adjusts the value of $Trigger$ to make the total HR still keep

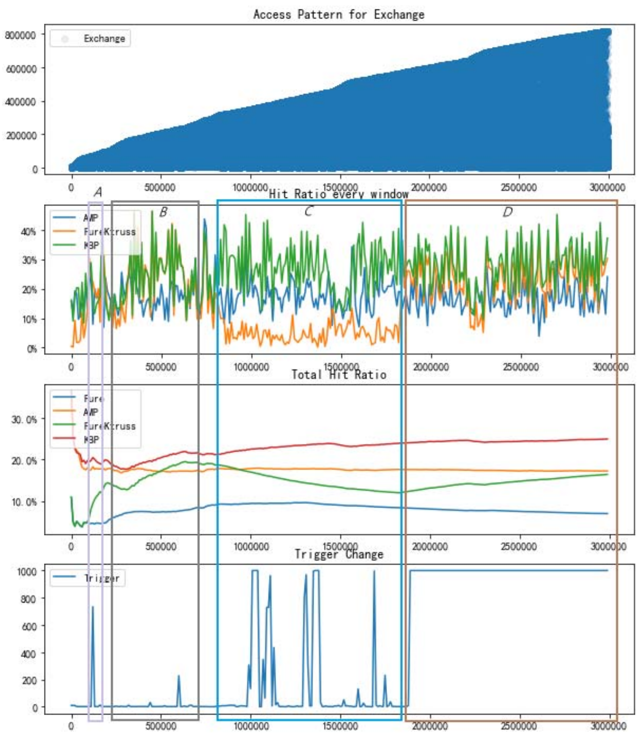


Figure 9. Effect of K-Truss Discovery And $Trigger$ on Exchange and the cache size is 128M. The first three plots are same meanings to Figure 8. The last one shows the change of $Trigger$ in KBP as I/O workload changes. The default value of trigger_max is 1000.

rising. This shows that the dynamic adjustment of $Trigger$ is effective. At D, KBP maximizes the value of $Trigger$ to reduce active prefetching. As a result, KBP still maintains an upward trend compared to the decline of AMP. At the same time, it can be seen that PureKTruss adapts to the new access pattern through the learning of the C process and presents a faster rising trend than KBP during D.

4.5. Effect of KBP Decay

In Figure 9, we can see that PureKtruss at C is decreasing continuously and in this process, more than 10 KBP decay has occurred in total. Finally, PureKtruss masters the new compound patterns so the overall HR is increasing in D. In this process, the operation of KBP decay can help PureKtruss adapt to new patterns more quickly. In addition, KBP decay can also effectively reduce the memory overhead.

4.6. Time and Space Overhead

The time cost of KBP mainly depends on compound pattern recognition, including the construction of EF and KTruss graph, and the query on KTruss graph. It is not difficult to know from the literature [10], [11], it takes less than 0.1ms to dynamically insert edges in the KTruss graph which is composed of hundreds of thousands of nodes, and query the max truss subgraph of a certain node. This is short compared to the I/O response time of the storage device (usually much greater than 1ms). In contrast, in algorithms such as C-Miner, it takes a lot of time to mine frequent subsequences so it cannot support real-time mining.

The space overhead of KBP is mainly EF and Ktruss graph. EF needs to record a lot of edges, while the occupation overhead of the Ktruss graph only needs $O(n)$, where n represents the nodes contained in the truss graph. As SA Filter is added to recognize sequential access patterns, a large number of records in EF are reduced. In addition, the KBP decay algorithm is also dynamically reducing the space occupied by KBP. In contrast, PureKtruss does not have SA Filter, so the space overhead will be relatively large. And in other probability-based methods such as MMs and Nexus-like, a probability graph needs to be maintained. As new blocks appear, space occupied gets larger.

5. Related Work

There are several ways to improve storage system cache performance. The first method is to propose a better cache replacement strategy (e.g., [14], [25], [26]). The importance of each block in the cache is evaluated through different indicators, including recency, frequency, and so on. Each replacement algorithm replaces the least important one and tries to keep the most important one in the cache. ARC [14], LeCar [25], and CACHEUS [26] which is the state-of-the-art caching replacement scheme are all for proposing more efficient replacement strategies.

The second method is to identify the access pattern [3] in the I/O stream, to fetch block from disk to the cache in advance when the storage system is idle.

The most common is the sequential prefetch strategy (e.g., [3], [16], [17]). For example, the well-known prefetching algorithm [16] in the Linux Kernel can efficiently deal with the pattern from a single sequential stream. Cloud environments, however, exhibit high levels of concurrency. This results in I/O workloads where multiple applications interleave I/O accesses that break the continuity of consecutive access patterns. Adaptive algorithms such as AMP [18] dynamically adjust the number of pages to be prefetched to prevent both cache pollution and prefetch wastage when the requests streams are interleaved, while TAP [17] uses a table to detect sequentiality and track longer history. Sequential prefetching has been widely deployed and commonly and widely used [27]. But the patterns covered by them are too narrow and they work well only for sequential workloads.

To obtain higher prediction accuracy, people begin to explore block access patterns based on history and propose general solutions integrating different types of patterns. One type of general algorithms is based on frequent subsequence algorithms (e.g., [2], [3], [4]). By using frequent sequence mining on the request sequence, we can obtain frequent subsequences which imply that the involved blocks are frequently accessed together in an access stream. C-Miner [3] and QuickMine [4] employ this technique but this type of algorithm has a large mining overhead and requires a large number of historical records for learning, and it is also not flexible.

Another type of general algorithms is to construct semantic graphs or probability graphs based on data block access patterns (e.g., [19], [20]). To obtain higher prediction accuracy, the algorithm based on the Markov model constructs a probability graph at runtime and can make predictions according to successors with high probabilities in the graphs. However, when the number of blocks is large, the space cost of the algorithm is large.

Our KBP efficiently reveals the correlation of blocks, and it is a highly adaptive online block access pattern mining strategy that can perform well on any workloads.

6. Conclusion

In this paper, we propose an online scheme, called KBP, to mine block access patterns for I/O prediction. KBP employs a novel architecture to coordinate the detection of different patterns, uses KBP decay to rapidly adapt to new workload and patterns, and adjusts the trigger step of active prefetching with online reinforcement learning to provide flexible prefetching. The experimental results demonstrate that KBP significantly outperforms traditional schemes.

Acknowledgments

We thanks the reviewers for their insightful feedback to improve this paper. This work was supported by National Natural Science Foundation of China (No.61872392, 61832020), Zhejiang Lab (NO.2021KC0AB04), Key-Area Research and Development Program of Guangdong Province (2019B010107001), Guangdong Natural Science Foundation (2018B030312002), Pearl River S & T Nova Program of Guangzhou (201906010008) and the Major Program of Guangdong Basic and Applied Research (2019B030302002).

References

[1] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[2] C. Zhu, F. Wang, and B. Hou, "Bpp: A realtime block access pattern mining scheme for i/o prediction," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[3] Z. Li, Z. Chen, S. M. Srinivasan, Y. Zhou *et al.*, "C-miner: Mining block correlations in storage systems." in *FAST*, vol. 4, 2004, pp. 173–186.

[4] G. Soundararajan, M. Mihailescu, and C. Amza, "Context-aware prefetching at the storage server." in *USENIX Annual Technical Conference*, 2008, pp. 377–390.

[5] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach." in *USENIX summer*, 1994, pp. 197–207.

[6] S. Yang, K. Srinivasan, K. Udayashankar, S. Krishnan, J. Feng, Y. Zhang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Tombolo: Performance enhancements for cloud storage gateways," in *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2016, pp. 1–14.

[7] Z. Li, Z. Chen, and Y. Zhou, "Mining block correlations to improve storage performance," *ACM Transactions on Storage (TOS)*, vol. 1, no. 2, pp. 213–245, 2005.

[8] T. M. Wong and J. Wilkes, "My cache or yours?: Making storage more exclusive," in *USENIX Annual Technical Conference, General Track*, 2002, pp. 161–175.

[9] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, no. 3.1, 2008.

[10] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1311–1322.

[11] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1298–1309, 2017.

[12] J. Wang and J. Cheng, "Truss decomposition in massive networks," *arXiv preprint arXiv:1205.6693*, 2012.

[13] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[14] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *Fast*, vol. 3, no. 2003, 2003, pp. 115–130.

[15] X. Yan, J. Han, and R. Afshar, "Clospan: Mining: Closed sequential patterns in large datasets," in *Proceedings of the 2003 SIAM international conference on data mining*. SIAM, 2003, pp. 166–177.

[16] F. Wu, "Sequential file prefetching in linux," in *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*. IGI Global, 2010, pp. 218–261.

[17] M. Li, E. Varki, S. Bhatia, and A. Merchant, "Tap: Table-based prefetching for storage caches." in *FAST*, vol. 8, 2008, pp. 1–16.

[18] B. S. Gill and L. A. D. Bathen, "Amp: Adaptive multi-stream prefetching in a shared cache." in *FAST*, vol. 7, no. 5, 2007, pp. 185–198.

[19] N. Tran and D. A. Reed, "Automatic arima time series modeling for adaptive i/o prefetching," *IEEE Transactions on parallel and distributed systems*, vol. 15, no. 4, pp. 362–377, 2004.

[20] J. Oly and D. A. Reed, "Markov model prediction of i/o requests for scientific applications," in *Proceedings of the 16th international conference on Supercomputing*, 2002, pp. 147–155.

[21] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *The VLDB Journal*, vol. 29, no. 1, pp. 353–392, 2020.

[22] D. J. Foster, A. Rakhlin, and K. Sridharan, "Adaptive online learning," *arXiv preprint arXiv:1508.05170*, 2015.

[23] A. Rakhlin, K. Sridharan, and A. Tewari, "Online learning: Beyond regret," in *Proceedings of the 24th Annual Conference on Learning Theory*. JMLR Workshop and Conference Proceedings, 2011, pp. 559–594.

[24] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[25] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, "Driving cache replacement with ml-based lecar," in *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[26] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with {CACHEUS}," in *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, 2021, pp. 341–354.

[27] J. Yang, R. Karimi, T. Sæmundsson, A. Wildani, and Y. Vigfusson, "Mithril: mining sporadic associations for cache prefetching," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 66–79.