LSP: A Locality-Aware Strip Prefetching Scheme for Striped Disk Array Systems with Concurrent Accesses

Xiaodong Shi
Computer College, Huazhong University of Science and Technology, Wuhan, Hubei 430074, PR China
Wuhan National Laboratory for Optoelectronics, Wuhan, Hubei 430074, PR China
Email: shixd.hust@gmail.com

Dan Feng
Computer College, Huazhong University of Science and Technology, Wuhan, Hubei 430074, PR China
Wuhan National Laboratory for Optoelectronics, Wuhan, Hubei 430074, PR China
Email: dfeng@hust.edu.cn

***Abstract*—In striped disk array systems, the independency of disks for prefetching is more important than parallelism under high concurrency of accesses, based on which strip prefetching with low read cost has more ability to improve the performance of RAID. However, it indiscriminately fetching all the involved strips limits its applicability. To solve this problem, we propose a Locality-aware Strip Prefetching Scheme (LSP), where it keeps track of the users' accesses and identifies the hot data areas. Only those strips located in the hot data areas will be prefetched and each prefetching request fetches one strip. LSP has several advantages. First, LSP adapts to the evolving workloads in an online and self-tuning fashion and satisfies the striped disk array systems due to the low read cost in each prefetching request. Second, LSP fully exploits the spatial locality in users' accesses. Here, the spatial locality is more general, which includes multiple simple patterns, such as loop references, sequentiality, reverse references, and other locality-awarded patterns. Third, LSP discriminates the hot data areas from the cold data areas when prefetching, which significantly alleviates the waste of disk bandwidth, optimizes the cache utilization and improves the prefetching accuracy. We have implemented the prototype of LSP algorithm in Linux kernel 2.6.18. The experimental results show that LSP outperforms SP and Sequential prefetching (SEQP) by up to 22.4% and 24.1% in terms of the average response time, and by up to 1.5 times and 2.3 times in terms of throughput, respectively.**

***Index Terms*— Striped disks array systems, Independency of disks, Strip prefetching, Spatial locality**

1. INTRODUCTION

Prefetching technologies are widely used in storage systems to bridge the performance gap between the processor and the storage device. Many prefetching schemes are dedicated to disk to break the performance bottleneck. For these schemes, one of the most important goals is to improve the throughput of disks by aggregating multiple contiguous blocks as a single request, where sequential prefetching is the most representative algorithm. For file-level prefetching schemes, multiple contiguous blocks fragmented from a file are also fetched in a single prefetching request. However, these approaches designed for single storage device cannot work well under the striped disk array systems [6, 7].

In striped disk array systems, the logically consecutive user data is split into multiple strips across distinct disks [1, 7, 8], as shown in figure 1. As a result, multiple requests located in distinct disks can be concurrently performed, which overlaps their service times. However, this splitting leads prefetching schemes to lose the ability to improve the disk throughput by aggregating large amount of contiguous blocks into a single prefetching request to save most of the disk seek time.

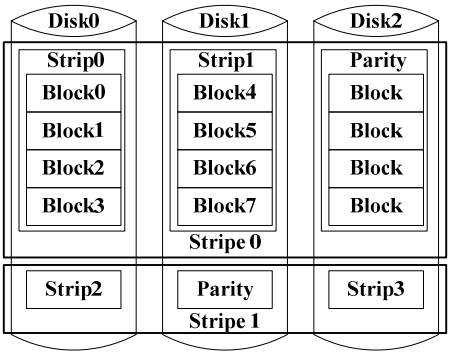

Fig. 1 The data organization in striped disk array

For prefetching under high accesses concurrency, the independency of disks is more important than the parallelism. The conventional prefetching schemes typically generate a large prefetch request consisting of multiple blocks that is not aligned in a strip. Therefore, a prefetching request may involve several strips. This induces the *independency loss* [2, 13] and significantly increases the prefetching cost in each involved disk, where the prefetching I/O is totally random with respect to the current disk head position. Moreover, the available sequentiality limited in strips cannot be fully exploited because the prefetching requests may involve only part of a strip. In contrast, the parallelism of prefetching has very limited ability to improve the performance of RAID under high execution concurrency because the striping based data

doi:10.4304/jcp.7.6.1303-1311

placement has fully exploited the parallelism among disks.

Most traditional prefetching schemes are designed for single device. Unlike them, *strip prefetching* is based on the striped disk array systems, where it prefetchs all the blocks in a strip whenever a block belonging to this strip is missed in the buffer cache.

As the importance of independency increasing, *strip prefetching* [2] with low read cost has more ability to improve the performance of striped disk array systems. Strip prefetching uses strips as the basic unit of prefetching. And hence, each prefetching request is dedicated to only one disk. As a result, the independency loss can be avoided and the sequentiality limited in strips can be fully exploited. Moreover, the prefetching request following the user request in the same disk has low read cost. However, strip prefetching doesn't consider the prediction accuracy or any access patterns in users' workloads, which are the most important factors in traditional prefetching schemes. This inaccurate prediction leads to two drawbacks: the cache utilization degradation and the disk bandwidth waste, which limit the applicability of strip prefetching.

Most I/O intensive applications exhibit spatial locality of accesses, which can be exploited to power the prefetching algorithms. Spatial locality means that if a block is accessed, the other blocks located in the same area are likely to be accessed in the near future. Many simple and regular access patterns are based on spatial locality, such as sequential reference, loop reference, reverse reference and stride reference et al.. Identifying the hot data areas can effectively exploit the spatial locality based access patterns and further benefit the corresponding applications. For example, the query service in database applications [19] is dominated by sequential requests, where blocks are placed continually and requested sequentially. Therefore, the hot data areas with spatial locality can be used to identify the query domains and benefit the database applications. There are some other types of applications exhibiting regular access patterns based on spatial locality, such as the loop references for science computation and reverse reference et al.. However, spatial locality isn't limited within regular access patterns, some seemingly random requests may also exhibit spatial locality. For example, a backend storage system for Web servers needs to supports amount of online clients. Although these clients have different access path, most of them may be interested in one or several domains, which leads to the spatial locality of referenced data. Therefore, an adaptive prefetching scheme that can identify the hot data area to exploit the spatial locality will benefit most applications that may exhibit different regular access patterns.

In this paper, we propose a *Locality-aware Strip Prefetching Scheme (LSP)*, where only those strips located in the hot data areas will be prefetched to exploit the spatial locality. In other words, LSP performs the prefetching only on those data areas with strong spatial locality. Due to spatial locality, these data areas are frequently accessed and become hot. By this way, LSP optimizes the strip prefetching with users' workloads characteristics. The spatial locality here is more general, which includes a set of regular access patterns, such as: sequential references, loop references, reverse references and stride references et al.. Unlike other prefetching schemes based on one regular access pattern (such as the sequential prefetching), LSP can work for as many workloads as possible. To the best of our knowledge, few approaches perform the block-level strip prefetching based on the hot data areas.

There are only a few studies considering the independency loss in striped disk array systems. Paek and Park [2] propose a dedicated cache replacement algorithm to alleviate the negative impact of strip prefetching, which works on the data prefetched by strip prefetching and improves the cache utilization. Moreover, SASEQP [13] tries to limit the depth of sequential prefetching into a strip to avoid the independency loss without considering the optimum depth of sequential prefetching itself. Different from these approaches, LSP exclusively focuses on strip prefetching and integrates the general spatial locality to improve its prediction accuracy. By this way, LSP can improve the cache utilization under cache replacement schemes as many as possible, and can significantly reduce the disk bandwidth waste.

The rest of this paper is organized as follows. Section II describes the prior work and the observations that motivate our work. LSP scheme is presented in section III in details. We analyze the performance of LSP through extensively trace-driven experiments in section IV. The conclusion is shown in section V.

II. Prior Work and Motivation

A. Prefetching Schemes for Storage Systems

a. File-Level Prefetching

The file-level prefetching schemes try to predict the next files in access stream through different approaches. Some approaches obtain the future information by exploiting the reference history [20], some are based on the hint of applications [21], and some are based on the hint from compiler [26]. However, all these schemes need the information about files, which is difficult to be implemented in low-level storage systems due to the narrow interface between them and file systems. Although these schemes can exploit the parallelism when they are used for striped disk array systems, the independency loss is obvious.

b. Block-Level Prefetching

There are many studies focusing on the block-level prefetching due to its transparency to file systems and storage applications. Some prefetching schemes predict the next blocks by exploiting the correlations between distinct blocks [14] based on the history references; some perform their prefetching requests based on one or multiple simple reference patterns. The sequential prefetching scheme is a representative algorithm that generates the prefetching requests based on the sequential reference pattern. Once a sequential stream is detected, they [3, 4, 5] aggregate large number of blocks into a prefetching request.

These prefetching schemes are popular and practical in

low-level storage systems. However, they are still designed for single storage devices without considering the characteristics in striped disk array systems, which results that traditional block-level prefetching schemes suffer from the independency loss.

c. Off-Line Prefetching

Some off-line studies focus on exploiting the parallelism of parallel disks systems under a single execution stream. *PC-OPT* [22] is an optimal algorithm in terms of cache utilization and disk parallelism. *Reverse aggressive prefetching* [9] designs an off-line algorithm that is used to serve the read-many reference strings. And, they prove that the algorithm of reverse aggressive is near optimal for typical parallel disks systems. Peter J. Varman etc. proposed other parallel prefetching algorithms [23, 24] for parallel disks systems, where they are based on the large scale blocks lookahead or off-line knowledge and try to fetch a block from each disk on every I/O period time. However, the complete future reference string used in these approaches is non-trivial to be achieved in practice.

d. Strip Prefetching

Strip prefetching is a simple prefetching scheme dedicated to striped disk array systems. There is a lack of studies for applicable strip prefetching. ASP [2] proposes a dedicated cache management (called adaptive cache culling) for strip prefetching, where it evicts (culls) prefetched and unused data at the earlier time in an adaptive manner to maximize the overall cache hit rate. The adaptive cache culling algorithm works on the data prefetched by strip prefetching. The buffer cache has to maintain and process those prefetched data before culling them. Moreover, a cache replacement scheme cannot influence the disk bandwidth waste of strip prefetching.

SASEQP [13] is a sequential prefetching algorithm, which limits the sequential prefetching depth into a strip. Similar as the strip prefetching, it can avoid the independency loss. However, this algorithm exclusively focuses on a simple reference pattern: sequential reference, while other patterns that can be exploited in strip prefetching are ignored. Moreover, the sequential prefetching schemes typically have their own models to determine the prefetching depth [3, 4] not limited within a strip.

The original strip prefetching has the advantage of avoiding the independency loss. However, it may not be an applicable prefetching scheme due to its inaccurate prediction, especially when there isn't any support from dedicated cache replacement. To address this problem, LSP not only retains the mentioned advantage, but also significantly improves the prediction accuracy by integrating the general spatial locality into prefetching algorithm.

B. Spatial Locality

The spatial locality is a common and inherent access pattern that significantly affects the throughput of storage systems. In web applications, 10% of files on web server workload contribute 90% of accesses and 90% of transferred data [17, 18]. More generally, the well-known principle called "The 80/20 Rule" [19] shows that 80% of accesses are always directed to 20% of files. Lots of studies focus on improving the spatial locality of accesses, which further enhances this trend. The studies [14, 15] realize the importance of spatial locality, thus they optimize the data layout by placing files or blocks with access correlations together. Arnan et al. [16] have demonstrated that reorganizing the data on disks to improve spatial locality can significantly improve the response time of I/O operation about 20-30% even using coarse grain volumes as atomic units. Most of popular file systems dynamically or statically place blocks on disks based on their correlations in access stream to reduce the I/O operation latency. However, these data may be placed in disk non-sequentially.

All these above mentioned studies illustrate that it is common for storage systems placing the hot data, frequently accessed within a comparable temporal locality, into a small area of a disk. Exploiting the hot data area can satisfy multiple simple reference patterns such as sequential reference, reverse reference, loop reference and stride reference. Therefore, introducing the hot data areas into strip prefetching can significantly improve the performance of RAID.

III. THE LSP SCHEME

Due to the shortcoming of conventional strip prefetching, we believe that the key to solve its problem is to efficiently identify the hot data areas. By this way, only those frequently accessed data with spatial locality will be prefetched.

The main idea behind LSP is to maintain and monitor the recently entered requests to evaluate whether there are data areas including those requests with spatial locality. Then, LSP performs the strip based prefetching in these data areas. More specifically, LSP maintains a preset size window for the recently entered requests. If there are multiple requests locating on the same data area, a new hot data area is identified and the size of this area is determined. When cache misses locate on any hot data area, the whole strip of blocks is prefetched.

It's important to note that the write requests can also achieve benefit from LSP. The hot data areas dependent on the accesses not only from read requests, but also from write requests. Therefore, the performance of write requests can also be improved due to the higher hit ratio without extra-overhead. Specifically, a write request typically includes three steps: read operation, modification operation and write operation. LSP can improve the write request performance through increasing the hit ratio of read operation. Combining our results with adaptive cache management and efficient write back policies for other operations may further improve the overall write requests performance, but it is beyond the scope of this work.

A. Identify the Hot Data Area

In LSP, the hot data area is fully and dynamically created and updated. At the beginning of LSP, there is no hot data area.

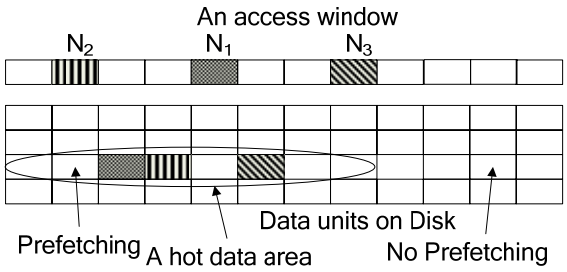

Fig. 2 Identifying the hot data area description

LSP maintains a window for the recently entered requests, whose size is set to 50 in this paper. Our experimental results show that too small size cannot fully exploit the available hot data areas, while too large size leads to the loss of potential prefetching requests (we describe this determination in section C). When the window is full, LSP traverses the window to search the blocks in distinct requests that have spatial locality. The spatial locality here isn't limited in the consecutive blocks. If the address gap between two data units from distinct requests isn't larger than 128KB (we describe this determination in section C) that is deemed to be optimum for workloads, we consider these two requests have spatial locality and locates at the same hot data area. If the requests with spatial locality are found, a corresponding hot data area is identified, just like the description in Figure 2.

We assume that a set of data units from requests with spatial locality, numbered $\{N_1, N_2,\ldots N_L\}$, are found, and hence its corresponding hot data area is identified with default length $2*(N_L-N_1)$, which ranges from $N_1-(N_L-N_1)/2$ to $N_L+(N_L-N_1)/2$. This default length includes three sub-areas: the head sub-area ranging $N_1-(N_L-N_1)/2$ to N_1, the middle sub-area ranging N_1 to N_L and the end sub-area ranging N_L to $N_L+(N_L-N_1)/2$. The head sub-area and the end sub-area may be accessed in the future, while the middle sub-area ranging N_1 to N_L has been accessed during identification. We set a larger size to a hot data area than middle sub-area, which can satisfy the future accesses. If we limit the default length into the middle sub-area, LSP will lag behind the accesses to this area and lose many potential prefetching requests. The default size of a hot data area is relative to the middle sub-area that will be dynamically adjusted according to the subsequent accesses, which can effectively avoid an excessive hot data area.

If a new request, whose involved blocks haven't been accessed before, is coming, the length of its corresponding hot data area is increased. If the prefetched data located in a hot data area is evicted from cache without any access, its length is reduced. If there are two adjacent hot data areas that have less address gap than 128KB, we merge them into one single hot data area. More specific description is shown in section B.

All the identified hot data areas will be stored in buffer cache. Due to the spatial locality of workloads, the number of hot data areas is far less than the disk space. Therefore, the space overhead in LSP is low.

B. LSP Algorithm

In this section, we describe the LSP algorithm as follows:

First, for each newly entering request, we check whether this request locates in an existing hot data area in the *identified hot data areas queue (IQ)*. If a hot data area is found, all blocks in the strip involved by this request are prefetched. If this strip is accessed for the first time, we identify this strip as *Accessed*, and the length of this area is increased by two strips in the direction of the middle sub-area extension. If no hot data area is found from IQ, the request is inserted into the LSP window for the future hot data area detection. In this step, the hot data areas in IQ are sorted by their start address, and hence traversing the IQ to search the hot data area can be CPU efficient.

```
/* Procedure to be invoked upon a new request */
Input: a new request i;
       IQ;
Output: an updated IQ;
a new request i is coming;
if (i ∈IQ)
{
    compute the strip (S) that includes the request i;
    Prefetch(S);
    if (S.accessed==false)
    {
      S.accessed=true;
      compute the hot-data-area (A) that includes S;
      A.length++;
    }
} else{
    insert request i into the window (W);
    W.length++;
    if (W.length>=threshold)
    {
      while (W.length>0)
      {
        remove a request j from the head of W;
        W.length--;
        if (there are another requests closing to j)
        {
          a new hot-data-area (Aj) is identified;
          insert Aj into IQ and sort Aj by its start address;
          if (there are Aj-1 or Aj+1 adjacent to Aj
             (distance<128KB))
             merge them;
        }
      }
    }
}

/*Procedure to be invoked upon an evicted block b*/
Input: an evicted block b;
       IQ;
Output: an updated IQ;
if (b is a prefetched not yet accessed block)
{
    compute the strip S that includes b;
    compute the hot-data-area A that includes S;
    remove S from A;
    A.length--;
    if (max(A.holes)>=256KB)
    {
      divide A into two hot-data-area;
    }
}
```

Fig. 3 The LSP algorithm description

Second, when the window is full, LSP checks the blocks in this window. If there are blocks accessed by distinct requests and spatially closing to each other, a new hot data area (A_i) is detected. This area is initiated by the default length and then inserted into the sorted IQ based on its start address. LSP checks the predecessor area (A_{i-1})

and successor area (A_{i+1}). If the areas are adjacent, LSP merges these two areas into one larger area. After all the available hot data areas in window are identified, the window is set to be empty.

Third, LSP keeps track of the evicted blocks from buffer cache. If any prefetched blocks in a strip are evicted without any access, we believe the hot data area including this strip has too large size and this strip has poor spatial locality. Therefore, LSP reduces the size of this hot data area and remove the strip from the current area. As a result, some 'holes' may exist in a hot data area. If the 'hole' is large enough, we divide an area into two hot data areas. In this paper, we set this threshold as 256KB that is described in section C.

In LSP, the size of hot data area is dynamically adjusted according to the user accesses. Therefore, it can efficiently reflect the characteristics of a specific workload. The LSP algorithm is described in Figure 3.

C. Implemention issues

In this section, we describe the implementation issues in details, which include the data structure in LSP and how to preset the parameters (thresholds).

LSP uses an access window to keep track of the recently referenced requests, based on which the evolving hot data areas can be obtains. Once a new request is coming, it is inserted into the window. If the window is full, a procedure for identifying hot data areas begins. Each time the identifying finishes, the whole window is emptied to be ready for a new round of identifying. This implies that the short-term history is more valuable than the long-term in our algorithm. Therefore, if the size of window is too large, the hot data areas can't be timely updated. Prefetching becomes ineffective because the detection is too late- the blocks in hot data areas have induced many misses. Moreover, large window also need more computing and cache resource that significantly increases the overhead of prefetching schemes. On the other hand, too small window size may lose some opportunities to detect the potential hot data areas. In our paper, we adopt the value 50 as the optimum window size because our experimental results in figure 11 demonstrate that LSP with this value achieves its best performance under most traces.

LSP uses a preset threshold to determine the upper bound size of holes in a hot data area. If a hole is larger than the threshold, its corresponding area is divided into two new areas. Therefore, the threshold size affects the total number of areas and holes in each area. Our experimental results show that if the size is small, the number of areas is dramatically increased, which will consume more computation and cache resources for prefetching. For each request, LSP has to traverse most of the areas to locate it. With the threshold size increasing, although the total number of areas may be reduced, more holes may be included in each area. For an entering request located in an identified area, LSP has to traverse all the holes in the area to determine whether the request falls into holes. If there are too many holes in an area, LSP becomes inefficient. To balance between the total number of areas and the number of holes per area, we adopt a threshold value 256KB that is optimum and adaptive to the workloads in our experiments (figure 12).

Another parameter in LSP is used to detect the hot data areas, which determines the maximum address gap between two requests that are considered to have spatial locality. In this paper, we set the value of this parameter as 128KB because we have very limited choices. First, this parameter should adopt a less value than the upper bound size of holes (256KB). Second, LSP is based on the strip prefetching, which means the strip size (64KB) is a basic unit in both prefetching and hot data areas detecting. Therefore, the address gap should be aligned in one or multiple strip. To satisfy both of the conditions, LSP adopts 128KB, which efficiency is also demonstrated in section IV (figure 13).

A key implemental detail for the LSP algorithm to work is to design an efficient data structure to locate existing hot data areas and holes. Without proper management, all the areas in IQ need to be traversed to locate a request, which increases the over head of LSP. To overcome the problem, we index the areas and holes using balance trees. Since the number of hot data areas for a specific workload is limited, only a reasonably large cache space is needed to maintain the indexing tree. Figure 4 shows the data structure for locating areas and holes. Each area is linked to a balance tree for indexing its potential holes. Meanwhile, all the areas are also indexed by a balance tree for efficient lookup. By this way, the overhead of LSP can be effectively reduced. And, the computation complexity for each lookup is limited into $O(log^M + Log^N)$, where M means the total number of hot data areas and N represents the number of holes in each area.

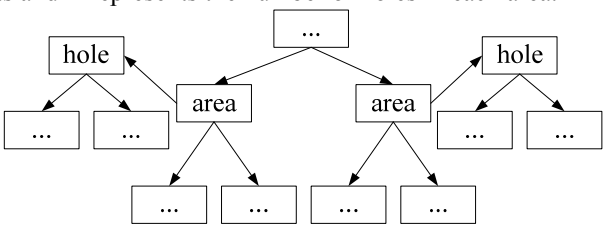

Fig. 4 The hot data areas and holes management data structure

IV. Performance Evaluation

We implement the LSP based on the Linux software RAID (MD), which is modified and embedded into the Linux kernel (Fedora Core 4 Linux, kernel version 2.6.11). The implementation has its own cache memory, whose space can be adjusted according to different needs of experiments (default 500MB). We name it as Linux software RAID with cache memory (LSR-CM), whose interface is the same as traditional MD.

To implement LSP, we track the I/O requests in the *make-request* function and issue its corresponding information to the recent access window. LSP can be activated when a request located in a hot data area is coming.

A. Experimental Setup and Methodology

The experiments used to evaluate the performance, are conducted on a server-class hardware platform with an Intel Xeon 3.0 GHz processor and 1 GB memory. We use 5 Seagate ST3250310AS SATA disks to construct the

RAID-5.

In our experiments, we compare the proposed LSP algorithm with the existing schemes *Strip Prefetching (SP)* and *Sequential Prefetching (SEQP)* [25], where the *SEQP-N* represents the maximum prefetching depth is *N*. We evaluate the performance of LSP through the extensive trace-driven experiments. To replay the traces on our storage system, we use the replaying tool based on RAIDmeter [10]. We use the traces from the Storage Performance Council [11], where the two types of workloads include OLTP and Web searching. For each type of workload, we randomly select three trace files, which are labeled as Fin[1-3] and Web[1-3]. The OLTP traces are characterized by the sequential access pattern, while the Web traces are more random.

To evaluate the impact of our work on throughput, we use synthetic workloads which can generate sufficient I/O requests per second. Specifically, we divide each trace into multiple equal fragments (sub-traces). The time stamps of all events in each fragment are equally shifted so that all the fragments start at the same time, like as show in [12]. RAIDmeter simultaneously replays multiple time-shifted fragments, which increases the I/O arrival rate while the practical scenarios based on the traces are kept. For example, the trace Fin1, collected from 10:00am to 11:00am, is divided into 10 fragments, where each fragment includes 6 minutes. The start time of each fragment is set to 10:00am and the time stamps of all events in this fragment are equally shifted. We denote the scale-up traces as Finx-n or Webx-n, where –n represent the number of fragments replayed simultaneously.

B. The Experimental Results

LSP can efficiently reduce the disk bandwidth waste and improves the prediction accuracy of strip prefetching. This advantage can be reflected in the average response time of RAID. Figure 5 shows the results about the response time of different prefetching schemes under Web traces. LSP obtains better performance than SP and SEQP 512 by up to 22.4% and 24.1%. We believe the performance improvement mostly derived from the higher cache utilization, because SP achieves a little performance benefit compared with SEQP512 that achieves the worst performance due to the random access characteristic in Web searching.

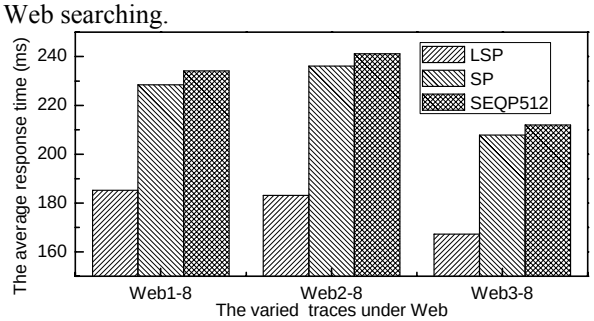

Fig. 5 The average response time of prefetching schemes under Web searching traces

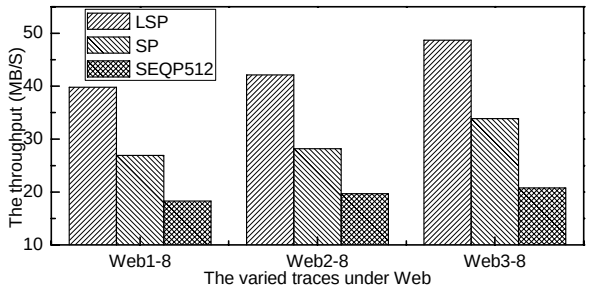

Fig. 6 The throughput of different prefetching schemes under Web searching

In figure 6, we show the throughput of different prefetching schemes. Under Web searching workloads, SEQP has very limited ability to improve the throughput of disks due to the random accesses. SP can improve the throughput of RAID because it avoids the independency loss and fully exploits the sequentiality in strips. However, the inaccurate prediction in SP limits the throughput improvement. As a result, LSP outperforms SP and SEQP by up to 49.2% and 2.3 times. This performance improvement demonstrates that LSP not only improves the cache utilization, but also optimizes the utilization of disk bandwidth.

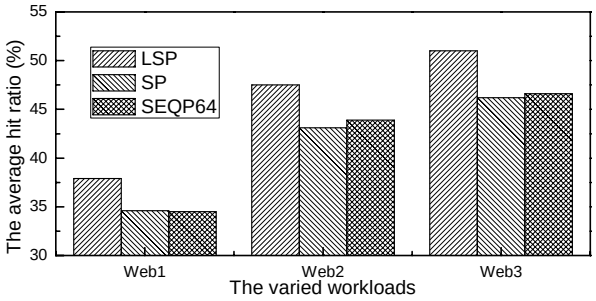

Fig. 7 The hit ratio of different prefetching schemes with different workloads

LSP can improve the prediction accuracy, which is more obviously reflected by the higher hit ratio rather than other performance factors, such as the average response time and throughput. Therefore, we evaluate the hit ratio of different prefetching schemes in figure 7. In this experiment, the SEQP64 compared with other sequential prefetching algorithms achieves the best hit ratio with the Web based traces. So, we adopt SEQP64 as a comparison algorithm. The experimental results show that LSP outperforms other prefetching schemes. Specifically, it improves the hit ratio by up to 10.4% and 9.9% compared with SP and SEQP64. For web traces with little sequentiality, sequential prefetching cannot efficiently exploit the spatial locality or improve the hit ratio of system.

C. Sensitivity Studies

Cache space is important for the efficiency of prefetching schemes. In this section, we evaluate the impact of cache space on prefetching schemes.

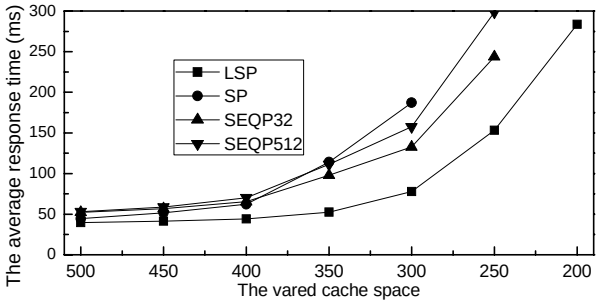

Fig. 8 The average response time of prefetching schemes with Web1-4 trace under varied cache space

Figure 8 shows the average response time of different prefetching schemes with Web1-4 workloads under varied cache space. With the cache space decreasing, the response time of all the prefetching schemes increase. However, this trend in LSP is slighter than that in others. When the cache space is reduced to 250MB, LSP outperforms SEQP32 and SEQP512 by up to 37.1% and 48.7%, which demonstrates that LSP can more efficiently utilize the buffer cache. The low performance of SP under small buffer cache further demonstrates the efficiency of LSP. When the buffer cache is sufficient, SP achieves better performance than both SEQP32 and SEQP512 because the buffer cache is large enough to store all the prefetched pages even under inaccurate prediction. With the buffer cache decreased, the inaccurate prediction in SP dominates the performance degradation, where SEQP32 outperforms SP by 29.1% under 300MB buffer cache. LSP significantly outperforms SP by 58.2% when the cache space is less than 300MB.

Figure 9 illustrates the throughput of different prefetching schemes with Web3-4 trace under varied cache space. LSP consistently outperforms other prefetching schemes. And, this advantage in LSP is more obvious when the cache space is small. For example, when the ache space arrives to 350MB, the throughput of LSP is higher than SP by a factor of 2.8 times. In contrast, this factor is only 1.4 times when the cache space is 500MB. Moreover, with the cache space decreasing, the performance of SP is degraded more seriously than SEQP, where SP outperforms SEQP32 and SEQP 512 by 38% and 38.6% when the buffer cache is 500MB. However, when the buffer cache is 200MB, SP only obtains 8.5% throughput improvement compared with SEQP32.

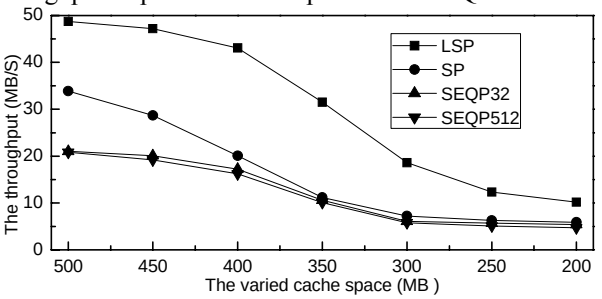

Fig. 9 The throughput of different prefetching schemes with Web3-4 under varied cache space

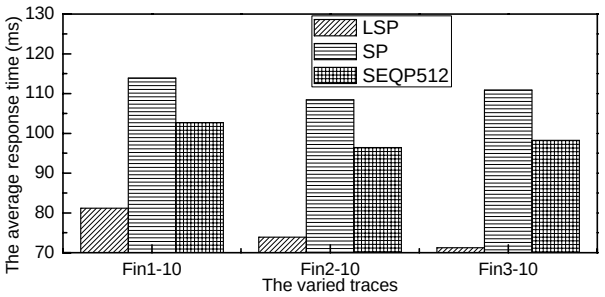

Fig. 10 The average response time under Fin2-10 (400MB default cache space)

LSP improves the prediction accuracy based on the spatial locality that includes multiple simple reference patterns, such as the sequential reference. To evaluate the efficiency of LSP in exploiting the reference patterns, we use the OLTP traces to perform the experiment, which generates many sequential accesses. In figure 10, LSP consistently outperforms both SP and SEQP512 by up to 35.8% and 27.5% under high concurrent execution. This demonstrates that LSP not only avoids the independency loss, but also efficiently exploits the sequential reference pattern in OTLP. In contrast, SP performs worse than SEQP512, where the average response time of SP is larger than that of SEQP by up to 11.4%. This illustrates that strip based prefetching without considering the access patterns of workloads cannot adapt to large scale storage systems although it can also avoid the independency loss.

D. *The impact of parameters*

In the implement to LSP, there are three important factors impacting on the performance and overhead of our algorithm. One of them is how to determine the size of recent access window, and another is to preset the upper bound size of holes in hot data areas. The third factor is the address gap used in hot data areas detecting.

The size of recent access window is a key factor that impacts the efficiency of hot data area detecting and the hit ratio of LSP. Therefore, LSP with the optimum window size should achieves the best performance with the highest hit ratio. To evaluate the optimum size, we conduct the experiments under different configurations shown in figure 11. The results demonstrate that when the window size ranges from 40 to 100, the highest hit ratio is obtained. However, larger window typically means heavier overhead in computation and space. Based on this consideration, we set the size of recent access window to 50.

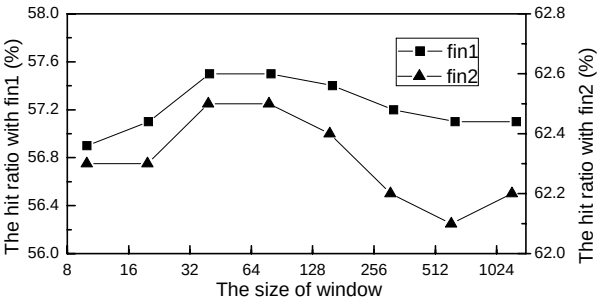

Fig. 11 The hit ratio with different window size and workloads

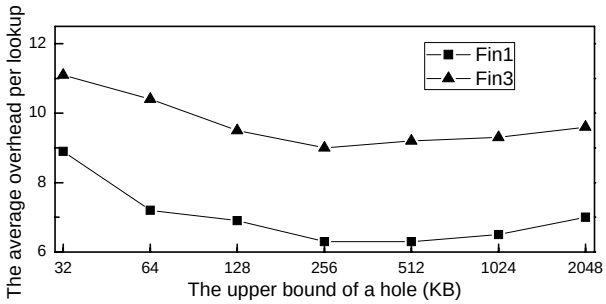

Fig. 12 The average overhead per lookup with different upper bound and workloads

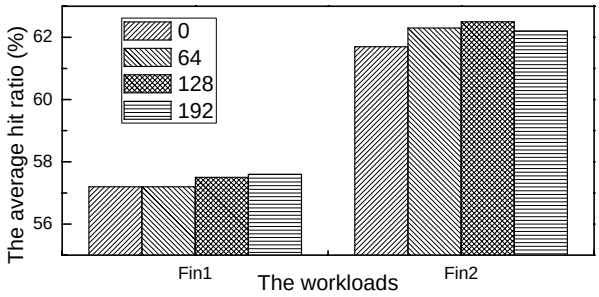

Fig. 13 The hit ratio with different address gap and workloads

The upper bound size of holes is another important parameter needed by LSP to balance between the total number of hot data areas and the number of holes per area. The preset value should minimize the sum of areas and holes involved in a single lookup, which can effectively reduce the prefetching overhead. Figure 12 shows the average computation overhead for each lookup with different upper bound size, through which the optimum size can be achieved. The results show that when the upper bound reaches 256KB, the average overhead for each lookup achieves its minimum value. With higher bound, the overhead has slight increase. In contrast, the lower bound leads to heavy overhead. The reason is that after the areas tree is searched, only parts of requests need to traverse the holes balance tree.

To evaluate the impact of different address gap on hot data areas detecting, we exam the hit ratio with all available configures. The results in figure 13 show that when the address gap is equal to 128KB, the hot data areas detecting achieves the highest efficiency. When the gap is zero, the detecting mechanism is similar to sequential prefetching, which has stricter limit in prefetching decision and some requests with spatial locality, but not sequentiality, may be ignored.

V. CONCLUSION

In this paper, we propose a locality-aware strip prefetching algorithm (LSP), which only performs the strip based prefetching on the hot data areas. LSP scheme can efficiently exploit the general spatial locality to work for workloads as many as possible, where multiple simple reference patterns (for example the sequential reference in this paper) can be used to improve the cache utilization and throughput of disks. The experimental results show that LSP outperforms SP and SEQP by up to 22.4% and 24.1% in terms of the average response time. LSP also achieves 1.5 times and 2.3 times throughput improvement over SP and SEQP under concurrent accesses. By integrating the spatial locality exploitation with the strip prefetching, LSP is adaptive to striped disk array systems and significantly improves both the cache utilization and the throughput of disk.

ACKNOWLEDGEMENT

This work is supported by the National Basic Research 973 Program of China under Grant No. 2011CB302301, 863 project 2009AA01A402, NSFC No.61025008, 60933002,60873028, Changjiang innovative group of Education of China No. IRT0725.

REFERENCE

[1] Santos, J.R., Muntz, R.R., Ribeiro-Neto, B., Comparing Random Data Allocation and Data Striping in Multimedia Servers. In Proceedings of the 2000 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, ACM Press, 2000, Santa Clara, CA, 44-55

[2] Baek, S.H., and Park, K.H. Prefetching with Adaptive Cache Culling for Striped Disk Arrays, In Proc. of USENIX Annual Technical Conference, pp. 363-376, June, 2008

[3] Liang, S., Jiang, S., and Zhang, X. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on (2007), pp. 64-.

[4] Li, M., Varki, E., Bhatia, S., and Merchant, A. Tap: Table-based prefetching for storage caches. In Proc. of the 6th USENIX Conference on File and Storage Technologies(Feb,2008), pp. 81-96.

[5] Gill, B.S., and Bathen, L.A.D. AMP: Adaptive multistream prefetching in a shared cache. In Proc. of USENIX 2007 Annual Technical Conference (2007), 5th USENIX Conference on File and Storage Technologies.

[6] D. Patterson, G.Gibson, and R.Katz. A case for redundant arrays of inexpensive disk(DISK). In International conferece on Management of Data, pages 10-116, June 1988

[7] P.Chen, E.Lee, G.Gibson, R.Katz, and D.Patterson. RAID: high-performance, Reliable secondary storage. ACM Computing Surveys, 26(2):145-188, June 1994

[8] P.Scheuermann, G.Weikum, P.Zabback: Data partitioning and load balancing in parallel disk systems, VLDB Journal 7(3), 1998

[9] Tracy Kimbrel and Anna R. Karlin. Near-optimal Parallel Prefetching and Caching. In Proceedings of the 1996 IEEE Symposium on Foundations of Computer Science, October 1996.

[10] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song. PRO: A Popularity-based Multi-threaded Reconstruction Optimization for RAID Structured Storage Systems. In FAST'07, Feb. 2007.

[11] Storage Performance Council. http://www.storageperformance.org/home.

[12] Y.Zhu and H. Jiang, "False rate analysis of Bloom filter replicas in distributed systems," International Conference on Parallel Processing(ICPP), pp.255-262, 2006.

[13] Baek, S.H., and Park, K.H., 2009. Striping-Aware Sequential prefetching for independency and Parallelism in Disk Arrays with Concurrent Accesses. IEEE ToC. 58(8). 1146-1152.

[14] Z.Li, Z.Chen, S.Srinivasan, and Y.Zhou. C-Miner: Mining block correlations in storage systems. In Third USENIX Symposium on File and Storage Technologies(FAST'04), pages 173-186, April 2004.

[15] P. Xia, D. Feng, H. Jiang, L. Tian, F. Wang. FARMER: A novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance. In the International Symposium on High Performance Distributed Computing(HPDC'08), June, 2008
[16] R.Arnan, E.Bachmat, T.K.Lam, and R.Michel. Dynamic data reallocation in disk arrays. ACM Transactions on Storage, 3(1), 2007
[17] L.Cherkasova and G.Ciardo. Characterizing temporal locality and its impact on web server performance. Technical Report HPL-2000-82, Hewlett Packard Laboratories, Jul. 2000
[18] L. Cherkasova and M.Gupta. Analysis of enterprise media server workloads: access patterns, locality, content evolution, and rates of change. IEEE/ACM Transactions on networking, 12(5):781-794, Oct. 2004.
[19] W.W.Hsu, A.J.Smith, and H.C.Yong, I/O reference behavior of Production Database Wrokloads and the TPC Benchmarks-an analysis at the logical level, ACM Trans. Database Syst. 26, No.1, 96-143 (March 2001)
[20] Peng Gu, Yifeng Zhu, Hong Jiang, Jun Wang: Nexus: A novel weighed-graph-based prefetching algorithm for Metadata Servers in petabyte-scale storage systems IEEE International Sympoium on Cluster Computing and the Grid(CCGRID'06), page:409-416, 2006
[21] R.H.Patterson, G.A.Gibson, E.Ginting, D.Stodolsky, and J.Zelenka, Informed prefetching and caching. In High Performance Mass Storage and Parallel I/O: Technologies and Applications. New York, NY:IEEE Computer Society Press and Wiley, 2001, 16, p. 224-244.
[22] KALLAHALLA, M., AND VARMAN, P. J. PC-OPT: Optimal offline prefetching and caching for parallel I/O systems. IEEE Trans. on Computers 51, 11 (Nov. 2002), 1333–1344.
[23] R. Shah, P. J. Varman, and J. S. Vitter. Online algorithms for prefetching and caching on parallel disks. In Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures, pages 255–264, June 2004.
[24] M. Kallahalla and P. J. Varman. Optimal read-once parallel disk scheduling. In Proceedings of the Sixth ACM Workshop on I/O in Parallel and Distributed Systems, pages 68–77, Atlanta, GA, 1999. ACM Press, New York, 1999.
[25] D.P. Bovet and M. Cesati. Understanding the linux kernel. O'Reilly, 2005
[26] J.Skeppstedt and M.Dubois, Compiler controlled prefetching for multiprocessors using low-overhead traps and prefetch engines. J. Parallel Distrib. Comput, vol.60, no.5, pp. 585-615, 2000.