Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

# Strip-oriented asynchronous prefetching for parallel disk systems*

Yang LIU[1,2], Jian-zhong HUANG[‡1,2], Xiao-dong SHI[2], Qiang CAO[1,2], Chang-sheng XIE[1,2]

([1]*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China*)

([2]*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*)

E-mail: yangl.hust@gmail.com; {husthjz, shixd, caoqiang, cs_xie}@mail.hust.edu.cn

Received Mar. 29, 2012; Revision accepted Sept. 12, 2012; Crosschecked Oct. 12, 2012

**Abstract:** Sequential prefetching schemes are widely employed in storage servers to mask disk latency and improve system throughput. However, existing schemes cannot benefit parallel disk systems as expected due to the fact that they ignore the distinct internal characteristics of the parallel disk system, in particular, data striping. Moreover, their aggressive prefetching pattern suffers from premature evictions and prolonged request latencies. In this paper, we propose a strip-oriented asynchronous prefetching (SoAP) technique, which is dedicated to the parallel disk system. It settles the above-mentioned problems by providing multiple novel features, e.g., enhanced prediction accuracy, adaptive prefetching strength, physical data layout awareness, and timely prefetching. To validate SoAP, we implement a prototype by modifying the software redundant arrays of inexpensive disks (RAID) under Linux. Experimental results demonstrate that SoAP can consistently offer improved average response time and throughput to the parallel disk system under non-random workloads compared with STEP, SP, ASP, and Linux-like SEQPs.

**Key words:** Parallel disk system, Strip, Sequential prefetching, Asynchronous scheduling

**doi:**10.1631/jzus.C1200085 **Document code:** A **CLC number:** TP333

## 1 Introduction

The large scale parallel disk system plays an important role in dealing with data intensive applications by providing both high throughput and large capacity. However, it remains a performance bottleneck in modern storage systems due to the long disk access latencies, causing execution to stall for milliseconds during a cache miss. The most efficient way to mask this latency is prefetching, where the required disk blocks are previously loaded into the cache before they are requested. Recently, considerable prefetching techniques have been studied in a variety of areas, such as processors (Kamruzzaman *et al.*, 2011), Web architectures (Lymberopoulos *et al.*, 2012), databases (Bowman and Salem, 2005), file systems (Zhang *et al.*, 2009), and storage controllers (Hartung, 2003). In this paper, we focus on the sequential prefetching technique for the parallel disk system, especially for the striped disk arrays.

In storage systems, the most popular prefetching technique is sequential prefetching, which predicts the future request patterns by sequential stream detection. The efficiency of sequential prefetching attributes to two factors, namely high prediction accuracy and low fetch cost. As sequential accesses are common in practical workloads, the future requests could be predicted by analyzing the recorded access information. That is, the more the access information recorded, the more accurate the prediction. Besides, file systems and databases tend to contiguously store the file data on disks.

‡ Corresponding author

* Project supported by the National Basic Research Program (973) of China (No. 2011CB302303), the National Natural Science Foundation of China (No. 60933002), and the Fundamental Research Funds for the Central Universities, China (Nos. 2012QN100 and 2011TUS-136)

©Zhejiang University and Springer-Verlag Berlin Heidelberg 2012

The sequential prefetching technique commonly performs aggressive large-size prefetching for sequential streams because the accesses to sequentially placed disk blocks can achieve a throughput an order of magnitude higher than can randomly placed disk blocks. By doing this, the high cost of disk seek latency can be amortized by a large amount of sequential blocks. This is why the prefetching cost is extremely low.

State-of-the-art sequential prefetching schemes such as STEP (Liang *et al.*, 2007), TAP (Li *et al.*, 2008), and AMP (Gill *et al.*, 2007) perform well in single disk systems. However, these schemes suffer from two limitations when they are applied to parallel disk systems. First, they neglect the most important characteristics of the parallel disk system, in particular, internal striping knowledge. As we know, the parallel disk system achieves uniform load balance across all disks by data striping, which splits the logical sequential user data into strips and then stripes them across the disks. Note that, the strip (RAID Advisory Board, 1999) is a set of continuous blocks residing in an individual disk and a stripe is divided into several strips. As a result, we can find that the parallelism nature of the parallel disk system is achieved at the expense of logical 'sequentiality loss'. For instance, when a prefetching request arrives in a RAID-5 system, the controller issues multiple I/O demands for the disks to concurrently serve it. Since the requested data is located across different disks, the prefetching request experiences multiple positioning time, thus increasing the prefetching cost. Therefore, a desirable prefetching scheme must employ the striping knowledge and adapt the prefetching operation to the physical data layout of the parallel disk system, thus exploiting the limited physical sequentiality.

Second, for conventional sequential prefetching techniques, prefetching strength becomes increasingly aggressive so as to amortize disk seek latency, and the requested blocks are required to be fetched synchronously. As a result, considering the relatively small prefetch cache, the already prefetched blocks are likely to be evicted before they are requested to make room for the new prefetching requests, which are known as 'premature evictions'. Moreover, it is not necessary to fetch all the blocks in a prefetching request simultaneously because the expected access time of the prefetched blocks is varying according to the arrival rate and the average request length of the corresponding sequential stream. So, a desirable prefetching scheme should give consideration to the access time difference of the prefetched blocks and load them asynchronously.

In this paper, we propose a strip-oriented asynchronous prefetching (SoAP) technique to improve the system latency and throughput of the parallel disk system. Different from other conventional sequential prefetching approaches, SoAP is dedicated to the parallel disk system and has several novel features. In summary, SoAP makes the following key contributions:

1. It uses a tool named the 'relationship graph' to predict the hit probability of a prefetching request for a given stream, and integrates this tool into a cost-benefit model to adapt the prefetching length to the system load as well as disk status. Such a model can significantly increase the prediction accuracy of prefetching and avoid the waste of idle disk bandwidth.

2. It provides a strip-oriented prefetching scheme, where the strip is used as the basic granularity for prefetching. In particular, all the prefetching requests must be aligned in the strip, and then split into multiple strip-level sub-requests, each of which contains a complete strip and is then delivered to an individual disk for scheduling. By doing this, prefetching operations are optimized according to the physical data layout of parallel disk systems, thus exploiting the maximum physical sequentiality.

3. It also proposes an asynchronous scheduling mechanism as a solution to resolve the aforementioned premature eviction. Specifically, the sub-requests of a prefetching request is not performed synchronously. Instead, each prefetching sub-request is associated with two timestamps, according to which the disk-level scheduling thread asynchronously and timely loads the corresponding strip into cache. This is also helpful in alleviating the bandwidth competition between user demands and prefetching requests.

We have implemented the SoAP algorithm in the software RAID module under Linux. Our results demonstrate that SoAP can deliver improvements in both average response time and throughput compared to conventional sequential prefetching schemes, hence proving that it is more efficient for the parallel disk system.

## 2 Background and motivation

In this section, we present the background knowledge and the observations that motivate our work in this paper.

### 2.1 Sequentiality and prefetching

Sequentiality is a common characteristic of practical I/O workloads, which accesses data contiguously. The ubiquity of sequentiality derives from the fact that file systems and databases tend to contiguously store file data on storage disks. Hence, many common file operations such as copy, scan, backup, and recovery lead to sequential accesses. Additionally, some important benchmarks exhibit strong sequentiality, such as TPC-D (Hsu *et al.*, 2001) and SPC-2 (Storage Performance Council, 2011).

A stream is a sequence of I/O requests of an application. It is considered to be sequential if the corresponding data is accessed contiguously. Due to the semantic gap between the file system and the storage sub-system, the only available information for sequential stream detection is logical block addressing (LBA). Therefore, sequential prefetching schemes can be easily deployed without any hint of applications or file systems. By mining the history of the past accesses, in particular the LBAs, the sequential prefetching schemes are able to achieve high prediction accuracy. Since they are less sophisticated than other kinds of forecasting techniques, these schemes are widely adopted in practical storage systems.

Existing sequential prefetching schemes are mainly divided into three classes, namely prefetch always (PA), prefetch on a miss (PoM), and prefetch on a hit (PoH) (Li *et al.*, 2008; Bhatia *et al.*, 2010). First, PA does not need a prediction module and always fetches contiguous data of a request. Assuming there is abundant cache space, it will achieve the highest hit rate but lowest efficiency. This is because a lot of prefetched data will never be accessed. In contrast, PoM only prefetches data on a miss, and thereby its miss rate is high. Additionally, the prefetching of random requests with little spatial locality will pollute the cache. PoH is popular in practice because it achieves both high hit rate and cache size economy. In PoH, a trigger mechanism (Gill *et al.*, 2007) is employed to avoid cache misses. Specifically, when PoH prefetches a set of disk blocks into the cache pages on a cache hit or miss, it chooses a trigger page by a 'trigger offset' before the end of the prefetched set. After that, when the trigger page is hit by a future request, a next prefetching request for this stream is activated.

### 2.2 Performance impact of sequentiality loss

Sequential prefetching schemes are mostly device-oriented, where prefetching operations should be issued with respect to the physical data layout. Unlike the single disk system, the parallel disk system has a distinct characteristic that the available physical sequentiality is limited in each strip. If a sequential I/O sequence exceeds the bounds of the strip, then the physical sequentiality is interrupted, which is known as 'sequentiality loss'.

Sequentiality loss is a major cause of the poor performance of conventional sequential prefetching schemes. Specifically, the service time of the disk consists of two parts: positioning time and transfer time. The ratio of transfer time to service time determines the bandwidth efficiency of the disks. If a prefetching request needs only one positioning operation, then it reasonably improves the bandwidth efficiency and reduces the cost of prefetching. Otherwise, if the prefetching request is performed in the parallel disk system where user data is striped, then it needs to consume more positioning time, and thus the prefetching cost increases significantly. As illustrated in Fig. 1, a prefetching request with eight blocks is issued by a sequential prefetching scheme and then separately performed under two different conditions: a single disk system or a parallel disk system consisting of two disks with the strip size of four blocks and the stripe size of eight blocks. The prefetching request consumes only one positioning time (Fig. 1a). In contrast, the prefetching request has to suffer from two positioning operations to transfer the same amount of data (Fig. 1b). Consequently, we could find that the bandwidth efficiency of the disks in the parallel disk system is much lower than that of the single disk system on account of sequentiality loss. To make matters worse, conventional prefetching algorithms use the block rather than the strip as the basic granularity of prefetching. As a result, the prefetching I/Os may involve only a part of a strip, which further incurs more disk traffic. This observation motivates our work in Section 3.2.3, where we use a strip-oriented scheme to adapt

the prefetching granularity to the physical data layout of the parallel disk system.

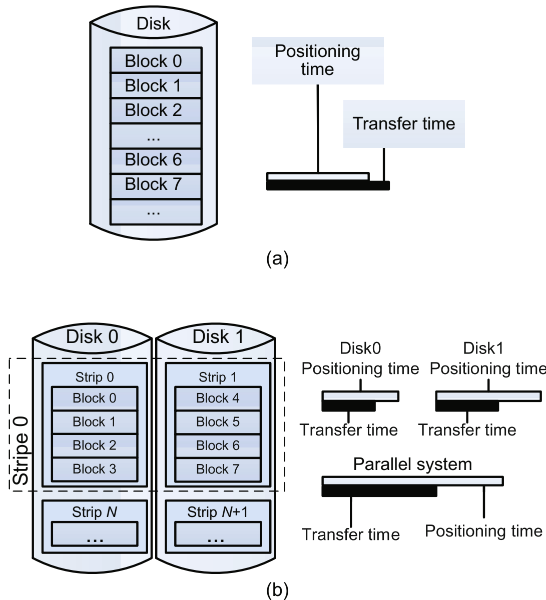


**Fig. 1 The effect of physical data layout on prefetching performance. (a) Performing the prefetching request (eight disk blocks) in a single disk system; (b) Performing the prefetching request (eight disk blocks) in a parallel disk system**

### 2.3 Performance impact of synchronous fetching

In parallel disk systems, the block set belonging to a prefetching request must be fetched synchronously with strict deadlines. For example, in a RAID system, the driver splits a prefetching request into multiple prefetching sub-requests and then delivers them to corresponding disks as soon as possible. After that, the thread handling these sub-requests keeps waiting until all the sub-requests are completed. During this process, all the involved disks serve these prefetching sub-requests with strict deadlines. This exemplifies what we call 'synchronous fetching'.

Synchronous fetching focuses on the parallelism of the executions of prefetching sub-requests to achieve fast data access. However, as sequential prefetching schemes are becoming increasingly aggressive in reducing the potential miss rate, the synchronous manner incurs several disadvantages. First, this will cause the above-mentioned premature eviction, where the prefetched data may be evicted before cache hit. Second, the average cost of misses is increased since the aggressive prefetching overloads the disks. Last, the average response time will be prolonged since bandwidth competition between prefetching and user requests increases the queuing length of the disks.

In practice, we observe that the prefetched blocks have different expected time for future accesses. It is related to the corresponding stream's attributes, such as request arrival rate and average request length. In addition, we find that prefetching cannot help when the system is overloaded. That is, absorbing the idle disk bandwidth is the most efficient way to improve prefetching. These observations motivate the work in this paper. In Section 3.3 we will show the proposed asynchronous scheduling scheme which asynchronously prefetches the desired blocks timely and economically but not immediately.

## 3 Design of the SoAP technique

The proposed SoAP technique is composed of three main modules. First, the sequential stream detection module is responsible for the detection of the sequential stream in the I/O workload. Second, the prefetching module is responsible for the generation and split of the prefetching request. It is the key module in SoAP, including novel features such as the leveraging of the relationship graph and the cost-benefit model. Last, the scheduling module is responsible for the scheduling of the prefetching sub-requests on the basis of disk. At the end of this section, the pseudocode of SoAP is described to put them all together.

### 3.1 Sequential stream detection

The role of the sequential stream detection module is to detect sequential access pattern from the workload. It serves two primary purposes: existing stream extension and new stream identification.

For the first purpose, a stream data structure is used to represent the run-time object associated with each identified sequential stream. This structure keeps attributes describing the corresponding stream, e.g., the timestamp of the most recent request, the average request length, the average inter-arrival rate, and the LBA of the next expected contiguous request. We define the LBA of the next expected request as 'trigger', which serves as the index

of the stream. All the streams are organized by a hash linked list named 'StreamQueue', using their triggers as the keys and the pointers of the stream structures as the values. When a new request arrives, the request address is searched in the hash table to see if any stream would be triggered. If so, it means an extension of a stream occurs. As a result, an update to the located stream is performed and a next prefetching operation is issued consequently.

For the second purpose, a particular address table referred to as 'HistoryTable' is used to record the I/O access history for further mining. When an incoming request does not extend any existing stream, its LBA is searched in HistoryTable. If an address is hit, then it suggests that two consecutive requests should be found and a new stream be created. After that, the hit address is required to be removed from the HistoryTable. Otherwise, if no address is hit, the LBA of the expected succeeding request is inserted into the HistoryTable for future sequential stream detection.

The sequential stream detection module involves both space and time overhead. To avoid excessive space overhead, the HistroyTable records only the LBA information of each request, which just occupies several bytes. In addition, two constant upper bounds are set to limit the number of streams in StreamQueue and the address entries in HistoryTable. Note that, LRU and FIFO are used in StreamQueue and HistoryTable respectively as the replacement policies. Considering that the StreamQueue and HistoryTable are both organized by the hash table, operations involving search and replacement can be completed in $O(1)$ time.

## 3.2 Prefetching module

The prefetching module will be activated when a new stream is detected or an existing stream is triggered. It determines the suitable prefetching length and the appropriate prefetching moment when generating a prefetching request. In this module, a relationship graph is employed to evaluate the hit probability of a prefetching request based on the sequentiality strength of the I/O workload. In addition, a cost-benefit model is proposed to facilitate the determination of the prefetching length. Finally, an algorithm splitting a prefetching request into strip-oriented sub-requests is provided to achieve the benefits from independency.

### 3.2.1 Relationship graph

Prior studies demonstrate that there is a strong correlation between the accessed length and the remaining length of a sequential stream. In addition, this correlation is stable for a given type of workload, such as workload of database (Hsu *et al.*, 2001) and networked storage server (Liang *et al.*, 2007). Inspired by these observations, we propose to use the relationship graph to mine this correlation. Depending on this graph, SoAP is allowed to evaluate the hit probability of a prefetching request to adjust the prefetching length.

As shown in Fig. 2, every vertex of the relationship graph represents a unique length, which is identified in units of strip. For example, a workload whose maximum stream length is $n$ will build $n$ vertices according to all the possible lengths. Each vertex has two fields: Len and Count. The former field represents the current length. The latter field specifies the count of streams whose lengths are equal to or larger than its Len field. The locality strength between the accessed length and the remaining length of a stream is represented as a weighted edge. For example, the degree of edge$(A, B)$ being 2 means that there have been two completed streams whose length has grown from 1 to 3. We use $P(x, y)$ to represent the hit probability of a prefetching request. In this function, $x$ represents the current stream length, and $y$ the prefetching length. Considering a stream whose current length and prefetching length are both 1, the hit probability of this prefetching operation is $P(1, 1) = \text{edge}(A, A)/A.\text{Count} = 40\%$; when the prefetching length increases to 3, the hit probability drops to $P(1, 3) = \text{edge}(A, C)/A.\text{Count} = 10\%$.

When a sequential stream is evicted from the StreamQueue, SoAP interprets that this stream has

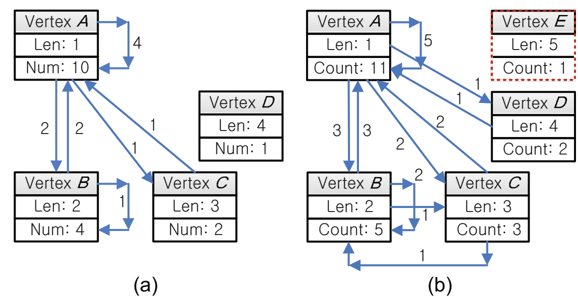


**Fig. 2 Construction of the relationship graph. (a) Original state of the relationship graph; (b) Updating the relationship graph with a sequential stream with length of 5**

been completed. Subsequently, the relationship graph should be updated by the length of this completed stream. An example is provided in Fig. 2 to illustrate the process of updating. The original graph in Fig. 2a consists of 10 sequential streams including 6 streams with length 1, 2 streams with length 2, 1 stream with length 3, and 1 stream with length 4. Subsequently, a sequential stream with length 5 is completed, which activates the graph updating procedure. In this procedure, one new vertex of $E$ should be built since the maximum stream length is 5. After that, every vertex whose Len field is less than 5 needs to increase the Len count by 1. Additionally, for any edge$(X, Y)$, if $X.\text{Len} + Y.\text{Len} \leq 5$, then the edge weight is increased by 1.

In practice, the overhead of maintaining a relationship graph is negligible in SoAP for two reasons. First, the update frequency of the relationship graph is relatively low since the update is activated only when a stream is completed. Second, there is an upper bound for the maximum number of the vertices in the relationship graph to avoid unnecessary overhead incurred by the vertex generations due to some extremely long streams, such as scan and flush. Assuming this constant upper bound is represented as $n$, it can be calculated that the main memory consumption is $4n^2$ bytes, and the time complexity of updating the relationship graph is $O(n^2)$ in the worst case. Considering that the strip size is large, the value of stream length normalized by strip is commonly much smaller than the upper bound $n$ in real workloads. Thereby, the cost of graph update is just low.

3.2.2 Cost-benefit model

The key issue for SoAP is to determine an optimal prefetching length. Without a proper strategy, the prefetching length can be either too conservative or too aggressive. In case of conservative prefetching, costly physical accesses to magnetic disks cannot be avoided because the prefetched blocks are too few. In case of aggressive prefetching, the prefetching suffers from inaccurate prediction, which incurs bandwidth waste, cache pollution, and premature eviction of prefetched blocks (Liang *et al.*, 2007). To address this problem, we exploit a cost-benefit model to make an acceptable tradeoff between prefetching length and the hit probability, thus maximizing the earning of prefetching.

In our model, the metric for cost-benefit evaluation is time. We use queuing theory to model the prefetching and disk system. The related notations are presented in Table 1. Specifically, disk service time includes seek time, rotation time, and transfer time. Considering the broken physical sequentiality in the parallel disk system, the rotation time has no direct relationship with the logical address distance between two requests. Thus, we refer to the time combining rotation time and data transfer time as track time (TT). The cost of prefetching can be represented as $\rho \times (\text{ST} + \text{TT})$, where $\rho$ represents the service intensity of the disk. It could be calculated as $\rho = \lambda/\mu$. After a prefetching sub-request is passed to the disk, there is a probability of $1-\rho$ to complete and hide the I/O time without disturbing the user requests. However, the arrival rate $\lambda$ and the service rate $\mu$ are difficult to derive. Therefore, SoAP assigns a queue for each disk to store the waiting user requests, and detects the queue length periodically to monitor the system load status. According to the average queue length $L$, we derive $\rho$ as $L/(L+1)$. Consequently, the cost and benefit of prefetching can be formulated as follows:

$$\begin{aligned} \text{Cost} &= \rho \cdot (\text{ST} + \text{TT}) \\ &= \frac{L}{L+1} \cdot (\text{ST} + \text{TT}). \end{aligned} \quad (1)$$

**Table 1 Notations for the cost-benefit model**

| Notation | Description |
|---|---|
| ST | Seek time |
| TT | Track time: rotation time and transfer time |
| $P$ | The probability of prefetching data requested |
| $\lambda$ | The arrival rate of requests |
| $\mu$ | The service rate of disk |
| $\rho$ | The service intensity of disk |
| $L$ | The length of request queue of disk |
| $S$ | A sequential stream |
| $S$.Len | The current length of a stream |
| $S$.avgRL | The average request length of a stream |
| $\text{EP}_{\text{len}}$ | The expected prefetching length |

The benefit of prefetching derives from the avoidance of the costly disk accesses. Assuming prediction accuracy is constant, the more aggressive the prefetching, the more the disk access it avoids. However, in practice, the hit probability decreases with increasing prefetching length. Since the hit probability could be queried from the relationship graph, the benefit and earning of a prefetching can

be formulated by Eqs. (2) and (3), where function $P(S.\mathrm{Len}, \mathrm{EP}_{\mathrm{len}})$ represents the hit probability of a prefetching request with prefetching length $\mathrm{EP}_{\mathrm{len}}$ in a stream whose current length is $S$.Len.

$$\mathrm{Benefit} = \left\lceil \frac{P(S.\mathrm{Len}, \mathrm{EP}_{\mathrm{len}}) \cdot \mathrm{EP}_{\mathrm{len}}}{S.\mathrm{avgRL}} \right\rceil \cdot (\mathrm{ST} + \mathrm{TT}), \tag{2}$$

$$\begin{aligned} \mathrm{Earning} &= \mathrm{Benefit} - \mathrm{Cost} \\ &= \left( \left\lceil \frac{P(S.\mathrm{Len}, \mathrm{EP}_{\mathrm{len}}) \cdot \mathrm{EP}_{\mathrm{len}}}{S.\mathrm{avgRL}} \right\rceil - \frac{L}{L+1} \right) \\ &\quad \cdot (\mathrm{ST} + \mathrm{TT}). \end{aligned} \tag{3}$$

To maximize the earning, we use an iterative method to determine the optimal prefetching length. Starting from $\mathrm{EP}_{\mathrm{len}} = 1$ with a step 1, a loop iterates to calculate the earnings of prefetching until $\mathrm{EP}_{\mathrm{len}}$ is equal to the upper bound $n$ of the relationship graph. By doing this, the prefetching length which brings the highest earning can be considered to be optimal.

### 3.2.3 Strip-oriented prefetching

Strip-oriented prefetching serves two main purposes. One is to adapt the prefetching to the physical data layout of the parallel disk system. The other is to determine when to prefetch blocks into the cache to avoid the next cache miss.

For the first purpose, SoAP breaks a prefetching request into multiple sub-requests, each of which contains a complete strip. That is, the base address of a sub-request must be aligned to a multiple of the strip size, and two sub-requests never overlap in the I/O address space. Fig. 3 shows a case example of splitting a prefetching request $A$ into three sub-requests. The rationale of choosing strip as the granularity of prefetching is that the maximum physical sequentiality of LBA is limited in the strip. Additionally, contiguous strips in a prefetching request are generally expected to be accessed in monotonically increasing time points by incoming user requests.

For the second purpose, when generating sub-requests, each of them is associated with two timestamps, namely expectedT and expiredT. The expectedT represents the expected time when the prefetched strip will be requested by future user demands. The expiredT represents the time threshold, beyond which the sub-request should be cancelled. Note that, 'Time' here refers to the logical time, which is measured by the number of total I/O accesses. Using Eq. (4), we derive the expectedT of the sub-requests in a prefetching request, where currentT is the current system time, SS represents the strip size, $S$.avgT represents the average arrival interval of the stream, and $i$ represents that it is the $i$th sub-request in the prefetching request.

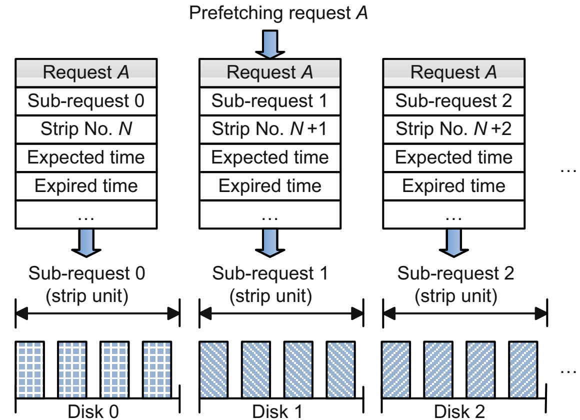


**Fig. 3 The procedure of splitting a prefetching request *A* into multiple sub-requests**

$$\mathrm{expectedT}_i = \mathrm{currentT} + \frac{2 \cdot i \cdot \mathrm{SS} \cdot S.\mathrm{avgT}}{\mathrm{EP}_{\mathrm{len}}}. \tag{4}$$

In Eq. (4), we set expectedT to twice the interval time to avoid premature degradation of the sub-request. The expiredT is set to five times the expectedT, where the fault cancelling rate of prefetching sub-requests is only 1%–3% in experiments with this setting.

With the increase of currentT, the sub-request with the least expectedT is believed to be the most urgent one that needs to be performed as soon as possible. This is because the less expectedT implies the sooner coming of a user request. However, once currentT exceeds expectedT, the prefetching sub-request becomes inactive because the sequential stream it belongs to is supposed to be complete.

To manage the sub-requests, SoAP associates each disk with three queues: a dispatching queue (referred to as 'D-Queue'), an active queue (referred to as 'A-Queue'), and an inactive queue (referred to as 'I-Queue'). D-Queue is used to store the user requests, A-Queue is used to store the arriving sub-requests whose expectedT is larger than currentT, while I-Queue is used to store the inactive sub-requests whose expectedT is exceeded by currentT.

Within these three queues, the pending I/O requests within them are sorted by the fields of LBA, expectedT, and expiredT, respectively. The newly generated sub-requests are always inserted into the A-Queues of corresponding disks in descending order of expectedT. To manage the three queues and the scheduling, each disk is associated with a monitoring thread (Fig. 4).

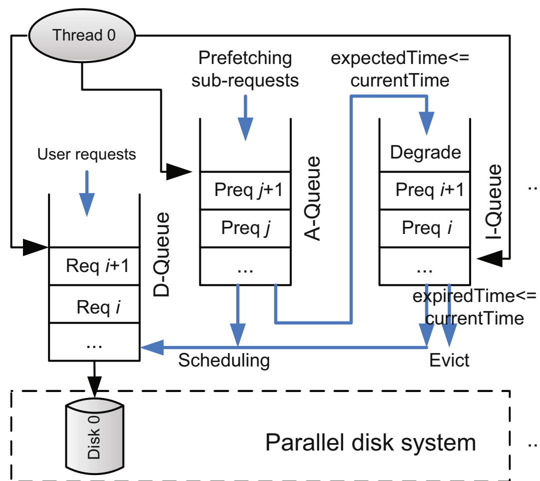


**Fig. 4 The management of sub-requests**

### 3.3 Asynchronous scheduling

Asynchronous scheduling involves several policies to determine the opportunities for performing sub-requests. They are chosen carefully to meet the strict requirements, such as being within the disk idle time or meeting certain spatial locality. Different from conventional prefetching schemes, the desired strips for a prefetching request are not synchronously fetched into cache. Instead, they are asynchronously loaded by the sub-requests according to three aspects: the arrival rate of the stream, load status of the disk, and the requirements of the scheduling opportunities.

Two main scheduling policies are designed in our implementation. The first is called 'contiguous scheduling'. Specifically, when a user request arrives in a disk on a cache miss, both A-Queue and I-Queue are searched to investigate if any sub-request has data intersection with it. If any is found, the sub-request will be inserted into D-Queue for emerging and performing.

Another policy called 'insertion scheduling' is illustrated in Fig. 5. The sub-request $M$ is inserted into the D-Queue because its LBA is between the LBAs of user requests $B$ and $C$. The rationale behind this policy is that the disk head moving in a straight line around the disk surface facilitates the reduction of expensive seek cost. Note that, SoAP searches only the A-Queue and finds the sub-requests with the least expectedT. Hence, it does not need to traverse the whole A-Queue because the queue has already been sorted by the expectedT.

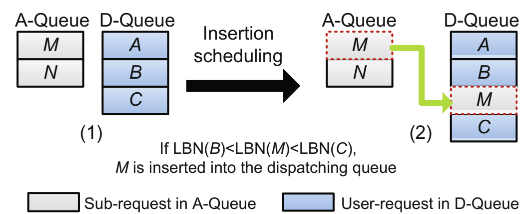


**Fig. 5 The insertion scheduling policy**

Update is required when there is an insertion in A-Queue or I-Queue. In this procedure of A-Queue, if a sub-request's expectedT is less than the currentT, then it is degraded into I-Queue. This degradation continues until the expectedT of the sub-request residing in the front of A-Queue is larger than the currentT. Similarly, for I-Queue, if a sub-request's expiredT is less than the currentT, then it will be evicted from I-Queue, which means this sub-request is cancelled. The eviction operation continues until the expiredT of the sub-request residing in the front of I-Queue is larger than currentT.

The monitoring thread periodically examines the instant length of the D-queue to examine whether the disk is idle. When D-queue's depth is not larger than 1, it is interpreted that the disk is idle. Then, the thread removes the sub-request in the front of A-Queue and inserts it into the D-Queue for execution. The chosen sub-request is considered to be the most urgent one in A-Queue, because it has the least expectedT. In this way, SoAP absorbs the idle bandwidth using sub-requests.

The advantages of asynchronous scheduling come from several aspects. First, it alleviates the premature evictions of the prefetched blocks, which is especially important due to the fact that the prefetching techniques are becoming increasingly aggressive. Second, it targets absorbing the idle bandwidth, thus minimizing the prefetching cost. Last, it makes disk access more sequential through the performing of scheduling policies.

It is important to make sure that the cost of asynchronous scheduling is low. To illustrate the related overhead of scheduling, we assume that the parallel disk system has $N_d$ disks, each of which has two queues to maintain the sub-requests. Therefore, the total main memory capacity consumed by these queues is $2N_dS_{sub}L_{avg}$ bytes in total, where $S_{sub}$ is the size of the sub-request, and $L_{avg}$ is the average number of pending sub-requests in a queue. It is obvious that the larger the $L_{avg}$, the larger the memory consumption. Fortunately, we find that $L_{avg}$ is generally less than 10 in our experiments. As a result, only tens of KB main memory is occupied to accommodate these queues. Similarly, as the depth of the queues is small, the computation overhead is negligible.

### 3.4 SoAP pseudocode

The pseudocode of SoAP is listed in Algorithms 1 and 2. Algorithm 1 focuses on handling the prefetching request generation, while Algorithm 2 addresses thread-level scheduling of sub-requests. We then describe how the SoAP algorithm performs.

In Algorithm 1, there are two important functions, PrefetchExecutor() and Prefetch(). Whenever a new request arrives, it should be handled by the function PrefetchExecutor(). First, the curT is increased by one at the start. After that, PrefetchExecutor() examines if the request generates a cache hit. If there is a cache hit, then this function traverses the streamQueue to check which stream is triggered. Consequently, a corresponding prefetching operation is issued. Otherwise, if this request generates a cache miss, then its predecessor request is searched in the HistoryTable. If it is found, then a new sequential stream is detected, which will be consequently recorded in the streamQueue. Subsequently, a prefetching request for this stream is issued. Otherwise, if there is no cache hit, then the request is inserted into the HistoryTable for future stream detection, and it is then served from the disk.

Function Prefetch() is activated when a new sequential stream is detected or an already detected sequential stream is triggered. To adaptively adjust the prefetching length to the load status, SoAP achieves the prefetchLength using the function CostBenefit(). By virtue of the relationship graph and the cost-benefit model, this function derives the optimum prefetching length. Consequently, the bounds of a prefetching request should be aligned in the strip. The trigger for the stream is also updated for the next prefetching operation. Finally, the prefetching request is split into multiple sub-requests and consequently delivered to the corresponding threads dedicated to the disks.

**Algorithm 1** Main prefetching procedure

```
PrefetchExecutor(req)
 1: currentT++
 2: if req ∈ Cache then
 3:   for any stream s_i ∈ StreamQueue do
 4:     if s_i is triggered by req then
 5:       UpdateStream(s_i, req)
 6:       Prefetch(s_i, req, triggerOffset)
 7:       break
 8:     end if
 9:   end for
10: else if req_pre ← HistoryTableHit(req, strideRange)
    then
11:   s_new ← NewStream(req_pre, req)
12:   Prefetch(s_new, req, triggerOffset)
13: else
14:   Fetch req from disks
15:   HistoryTableInsert(req)
16: end if

HistoryTableHit(req,strideRange)
 1: for any r ∈ RequestTable do
 2:   if req.Addr ∈ [r.Addr + r.Len, r.Addr + r.Len +
      strideRange] then
 3:     Remove r from HistoryTable
 4:     return r
 5:   end if
 6: end for
 7: return NULL

NewStream(req_pre, req)
 1: s ← new stream
 2: s_evict ← EnQueue(StreamQueue, s)
 3: UpdateRelationGraph(s_evict.len)

Prefetch(s, req, trigOff)
 1: prefetchLength ← CostBenefit(s) × stripeSize
 2: endAddr ← req.Addr + req.Len + prefetchLength − 1
 3: s.Trigger ← endAddr − trigOff
 4: endAddr = endAddr + stripSize − endAddr mod
    stripSize
 5: for any Sub-request_i ∈ [req.Addr, endAddr] do
 6:   compute its expectedT and expiredT
 7:   deliver Sub-request_i to corresponding disk
 8: end for
```

**Algorithm 2** Asynchronous scheduling procedure

```
ThreadScheduler(req)
 1: while A-Queue.Front.expectedT ≤ currentT do
 2:    req=DeQueue(A-Queue)
 3:    Insert(I-Queue, req)
 4: end while
 5: while I-Queue.Front.expiredT ≤ currentT do
 6:    DeQueue(I-Queue)
 7: end while
 8: if req is a prefetching sub-request then
 9:    Insert(A-Queue, req)
10: else if req is a user request then
11:    Insert(D-Queue, req)
12:    for any Sub-request_i ∈ A-Queue or I-Queue do
13:       if Sub-req_i ∩ req ≠ NULL then
14:          Insert(D-Queue, Sub-req_i)
15:       end if
16:    end for
17:    if A-Queue.front ⊆ [D-Queue.front.LBA,
       D-Queue.rear.LBA] then
18:       Insert(D-Queue, A-Queue.front)
19:    end if
20:    if D-Queue.depth ≤ 1 then
21:       Insert(D-Queue, A-Queue.front)
22:    end if
23: end if
```

Algorithm 2 provides the details of per-disk thread-level scheduling for user I/Os and prefetching sub-requests. Whenever a sub-request arrives in a disk, it is handled by ThreadScheduler(). First of all, the A-Queue and I-Queue should be updated to degrade the inactive sub-requests from A-Queue and to cancel the time-out sub-requests from I-Queue. After that, if the arriving request is a prefetching sub-request, then it is inserted into the rear of A-Queue. Otherwise, if it is a user I/O, then continuous scheduling will be used for A-Queue and I-Queue, and insertion scheduling will be applied for A-Queue to exploit spatial locality. Additionally, if the depth of D-Queue is less than or equal to 1, then it means the disk is or will be idle. Consequently, the prefetching sub-request in the front of A-Queue, which is most urgent, will be inserted into D-Queue.

## 4 Experimental methodology

We have implemented a prototype of SoAP based on Linux of kernel version 2.6.11. Several modules have been added in the Linux Software RAID (MD) to support the proposed scheme. The default cache size is 512 MB and the adopted replacement policy is LRU. To evaluate our design, we conducted the experiments on a 3.0 GHz Intel Xeon computer with 1 GB RAM and 5 Seagate ST3250310AS SATA disks by default, each of which has 250 GB capacity, 7200 r/min rotation speed, and 8.5 ms average seeking time. We configure these disks as RAID-5 in our experiments. Note that a small portion of the disks' capacity has been used for holding workload traces and trace replaying tools.

The evaluation of our design is driven by a tool called RAIDmeter (Tian *et al.*, 2007). It deals with the replaying of traces in the environment of parallel disk systems. In detail, RAIDmeter reads the I/O operations from traces and then creates corresponding requests. After that, these requests are passed to the block storage devices according to their timestamps.

In our experiments, two types of traces are used, namely Financial and Web Searching. Both of them derive from Storage Performance Council (2011). Financial traces are characterized by the sequential access pattern because they were obtained from online transaction processing (OLTP) applications running at a large financial institution. Unlike Financial traces, Web traces were collected from the searching engine workload, which is more random although it exhibits some spatial locality in the access pattern. For each type of workload, we randomly select parts of the trace files, which are labelled as Fin1–2 and Web1–3. To evaluate the performance under heavy load, we decompose a trace into multiple sub-traces and set their start time as the same time to generate sufficient I/O requests. These scaled up traces are denoted as Fin$x$-$n$ or Web$x$-$n$, where $n$ represents the number of sub-traces replayed simultaneously.

We also use the benchmark IOzone to simulate the forward sequential reads. IOzone is a file system benchmark to measure a variety of file operations, such as sequential and random read, write, create, and delete. In our experiment, it is set with a varied number of concurrent processes ranging from 1 to 20. Moreover, its address space is limited within a fixed file size of 4 GB, and the block size is set to 4 KB.

We compare SoAP with existing techniques, including SEQP (Bovet *et al.*, 2005), SP, STEP (Liang *et al.*, 2007), and ASP (Baek and Park, 2008) with respect to the key metrics of response time and throughput. SEQP is similar to the Linux's default

sequential prefetching algorithm in the kernel. We vary the upper bound of the prefetching request size (prefetching threshold) in SEQP to achieve its best performance. SP is a scheme that always prefetches all blocks of the strip to which the requested blocks belong on a cache miss. Although SP is designed for parallel disk systems, it involves several problems such as poor prediction accuracy and bandwidth waste. STEP is a recently proposed sequential prefetching algorithm. It takes advantage of the sequentiality in disk accesses to maximize the earnings. ASP is similar to SP, except that it adopts a culling cache management scheme that culls the unrequested prefetched blocks earlier. It also has an online cost estimation model that deactivates strip prefetching when the hit rate of ASP is lower than that of no prefetching.

## 5 Performance evaluation and discussions

In this section, we present a number of measurements illustrating the performance improvements with SoAP.

### 5.1 Real-world trace performance

In Fig. 6, we examine the performance of various prefetching techniques in terms of average response time using real traces, including Financial and Web Searching. In case of sequential Financial traces, SoAP is convincingly the best as expected. It outperforms STEP by 32.8%, SEQPs by 22.3%–43.6%, ASP by 43%, and SP by 100%. Among all these techniques, SP performs the worst because it always fetches a full strip on a cache miss and does not have any prediction module. ASP performs better than SP due to the contribution from culling cache management which evicts the unrequested prefetched blocks earlier. However, it still fails to solve the problem of low prefetching accuracy, thus wasting the precious bandwidth. STEP performs better than SP because it determines the prefetching length based on a cost-benefit model. However, this model is designed for the single disk system instead of the parallel disk system. Note that the SEQPs have varied upper bounds for prefetching length. The results show that the optimum bound should be around 128 KB.

For random Web Searching traces, all the prefetching algorithms perform similarly, because there are few sequential streams supporting the prefetching. Even so, SoAP still performs better than others due to its cost-benefit model. The relationship graph provides a low hit probability for prefetching, thus deactivating the prefetching in SoAP. ASP performs almost the same as SoAP because it has an online cost estimation model that deactivates strip prefetching when the hit rate of ASP is lower than that of no prefetching.

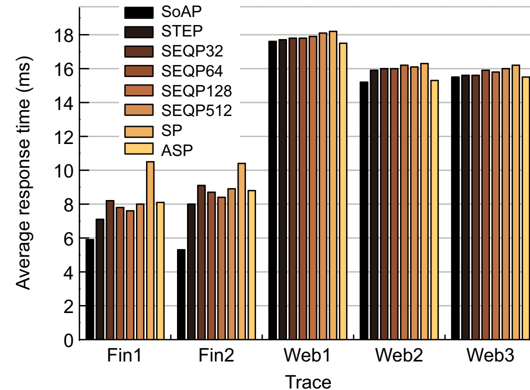


**Fig. 6 The average response time test with different prefetching strategies and workloads**

Fig. 7 provides the throughput comparison between SoAP and other prefetching schemes using two sets of traces. For Financial traces, we can see that the throughput of SoAP is consistently higher than those of other schemes. Specifically, SoAP outperforms SEQP32 by 100%, and SEQP512 by 71.1%. Meanwhile, it achieves up to 29.4% and 53.6% throughput improvements over ASP and SP, respectively. Since Financial traces are considered to be composed of multiple interleaved sequential streams, these results demonstrate SoAP's efficiency in improving sequential accesses. Note that the throughput of SEQP32 is the lowest. We infer that it suffers from the small upper bound of prefetching depth, which might induce much more disk I/Os than other schemes, especially under workloads with high sequentiality.

Even for Web Searching workload, which is somewhat unfair for SoAP since there is little sequentiality to employ, SoAP still achieves better throughput than STEPs and SP. However, we can see that ASP outperforms SoAP by 1.6%–4.7%. Although it is marginal, we still need to explain why this occurs. The first reason is that ASP will stop strip-prefetching in the random workloads. In contrast,

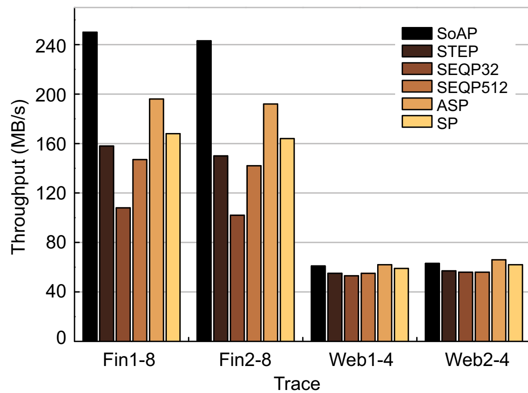


**Fig. 7 The throughput test with different prefetching strategies and workloads**

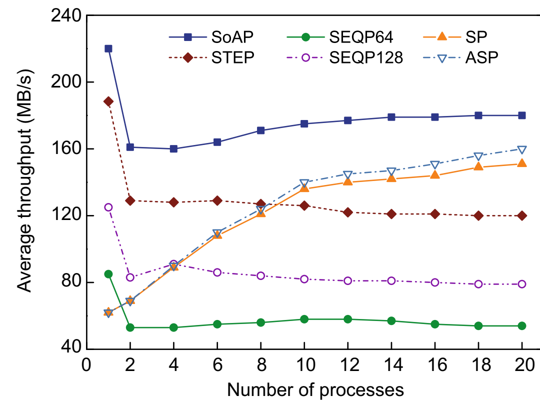


**Fig. 8 The throughput test with different prefetching strategies and benchmark Iometer**

SoAP still tries to prefetch the small streams in Web workload, which benefits the response time but at the cost of throughput. The other reason is that SoAP consumes extra main memory for stream detections, which do not benefit the performance under random workloads.

## 5.2 Benchmark performance

Fig. 8 depicts the aggregate throughput achieved by different prefetching schemes under workloads generated by benchmark Iometer. We set the strip size of RAID-5 as 64 KB and vary the number of concurrent replay processes from 1 to 20, each of which generates forward reads. The key observation is that strip-ware prefetching schemes are more suitable for high concurrency. We can see that, in case of fewer than eight concurrent processes, STEP performs better than SP and ASP. When the count of concurrent processes is more than eight, the strip-aware algorithms perform better than all strip-unaware algorithms, such as STEP and SEQPs. The reason why STEP is more efficient than SP and ASP under low concurrency is that STEP can perform more aggressive prefetching for sequential streams with high confidence than SP and ASP, whose prefetching upper bound is a consistent strip. However, when the concurrency of workload is high, the prefetched data of aggressive prefetching may be prematurely evicted from cache before they are requested. Besides, the prefetching requests' bounds of STEP are not aligned in the strip, thus incurring more disk traffic. As expected, SoAP is universally the best under different numbers of concurrent processes. This is due to the fact that SoAP is a prefetching technique which is stream-aware, physical data layout aware, and prefetching-in-time.

Although SP, ASP, and SoAP are all strip-oriented prefetching techniques, SoAP outperforms SP by 64.2% and ASP by 60.7% on average. The performance disparities are owing to the large differences between them. First, SoAP is a sequential prefetching scheme with a stream detection module to generate the prefetching request, and a cost-benefit model to determine the prefetching length. In contrast, SP and ASP have no prediction module at all. They focus on the simplicity and low overhead at the expense of poor prediction accuracy and high miss rate. This is why ASP uses adaptive cache culling (ACC) to evict the unrequested prefetched data in time. Thus, they cannot settle the problem of premature eviction, and hence the ghost strip cache has to be implemented. Second, the prefetching length upper-bound of ASP and SP is the strip size. Even after incorporating the MSP, the prefetching granularity of ASP is neither the strip size nor the stripe size. In contrast, the prefetching length of SoAP is adjustable and can be optimized as any multiple of the stripe size to obtain the maximum benefit. Last, SoAP is implemented as a prefetch-on-hit scheme, where there is only one cache miss for all the requests in a sequential stream. Unlike it, ASP and SP are prefetch-on-miss techniques, thus suffering a high miss rate in practice.

## 5.3 Scalability study

For parallel disk systems, system scalability is an important factor that directly relates to system performance. We study the scalability of the parallel disk system running various prefetching schemes by changing the numbers of trace replay processes and

disks.

In Fig. 9, we examine the system average response time under various prefetching schemes with five disks while the number of trace replay processes increases from 1 to 16. The system average response time increases when the number of replay processes increases. This is because the larger the number of trace replay processes, the heavier the system loads. Note that SoAP scales much better than other prefetching algorithms because its cost-benefit model is workload adaptive. Specifically, it outperforms SEQPs by 21.5%–47.1%, STEP by 16.9%–43.4%, SP by 24.9%–58.4%, and ASP by 9.7%–31.9%.

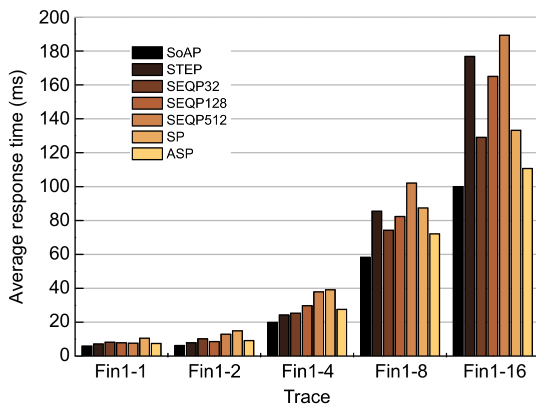


**Fig. 9 The average response time test with an increasing arrival rate and a constant disk number of five**

The number of disks in a parallel disk system is especially important for system scalability since it determines the maximum available parallelism. In Figs. 10a and 10b, we examine the system average response time under various prefetching schemes with a constant number of trace replay processes while the number of disks increases from 3 to 8. These disks are organized as RAID-5 with the strip unit size of 64 KB. We can observe that for all these schemes the average response time decreases when the number of disks increases. However, SoAP is more efficient and scalable than other schemes under both light and heavy load, because it could adapt the prefetching length according to the disks' load status, thus absorbing the idle bandwidth for prefetching.

Figs. 10c and 10d show the increase of throughput when the number of disks increases from 3 to 8 under workload Fin1-8 and Fin2-8. We can see that the increasing number of disks provides speedup in system throughput. It is clear that SoAP is more

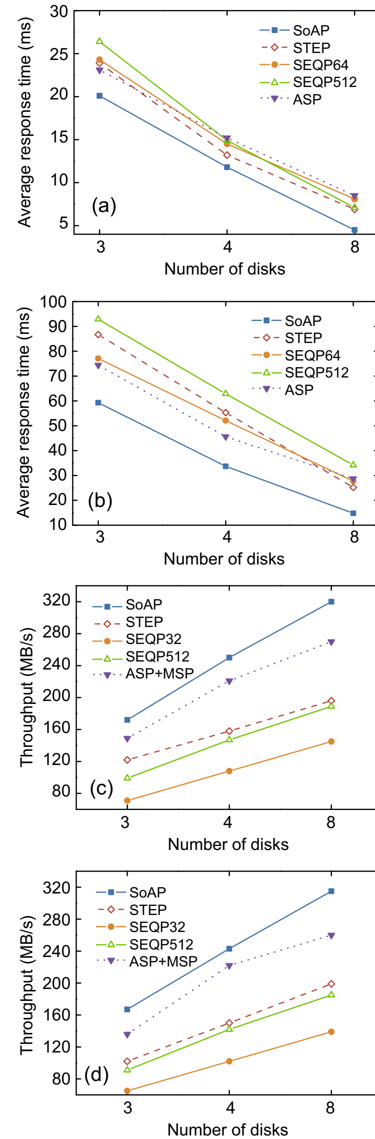


**Fig. 10 The performance test with a varied disk number. (a) Fin1-1; (b) Fin1-4; (c) Fin1-8; (d) Fin2-8**

scalable than other schemes. Although ASP+MSP exhibits most close performance, SoAP still outperforms it by 15.4%–18.5% under Fin1-8, and 9.4%–22.8% under Fin2-8.

## 5.4 Sensitivity study

In this subsection, we evaluate the impacts of the strip size on system performance using different prefetching schemes. The strip size determines the maximum physical sequentiality, thus impacting the performance of prefetching schemes. To evaluate the sensitivity of SoAP to the strip size, we set the experiment platform with a varied strip size and a

constant disk number of five in Fig. 11.

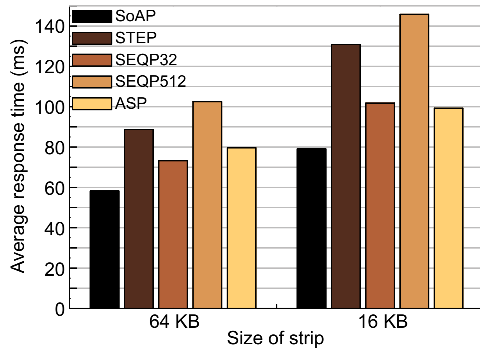


**Fig. 11 The average response time test with a varied strip size and a constant disk number of five under Fin1-8**

The results show that the average response time of all prefetching schemes changes when the strip size decreases from 64 KB to 16 KB. This is because the parallel disk system with a small strip size achieves higher parallelism but more sequentiality loss. As a result, the same length prefetching request incurs more disks' accesses than using a large strip size, hence increasing the cost of prefetching. Under workload Fin1-8, we can see that SoAP is more adaptive to the change of the strip size, and it consistently outperforms other algorithms. Specifically, SoAP outperforms STEP by 34%, SEQP512 by 43.2%, and ASP by 26.4% with 64 KB strip size. Even with 16 KB strip unit, SoAP achieves better response time than STEP by 39.3%, SEQP512 by 45.7%, and ASP by 20.3%. The reason is that SoAP is designed to adapt the physical data layout by virtue of strip-oriented prefetching and it could flexibly adjust the prefetching length according to the cost-benefit model.

## 5.5 Cost-benefit model

It is very necessary to understand how the cost-benefit model affects the prefetching in SoAP, because this model plays an important role in determining the optimal prefetching length according to the load status. In Fig. 12, we examine the average prefetching length of several prefetching schemes under increasing system load. We set the strip size of RAID-5 as 16 KB and vary the number of concurrent replay processes from 1 to 10. SoAPnoM represents the SoAP prefetching without disk status monitoring. We can see that the average prefetching length of SoAPnoM and SEQPs fluctuates slightly, and the trend is to level off. It indicates that these schemes are not sensitive to the workload. In contrast, the prefetching length of SoAP has a clear relationship with the system load. With the load increasing, the prefetching length decreases. This is because there will be less idle bandwidth when the disk traffic is heavy, and prefetching will not help when the system is overloaded.

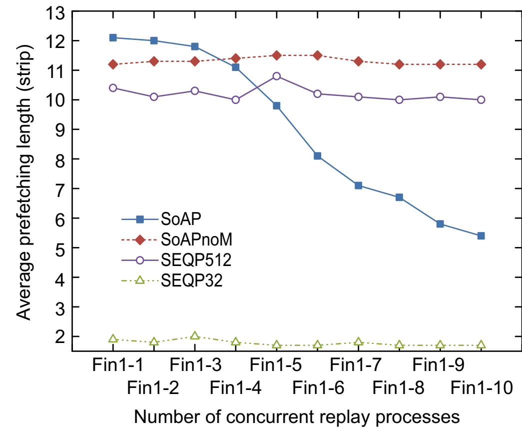


**Fig. 12 The average prefetching length test with different prefetching strategies (the strip size of RAID-5 is set to 16 KB)**

## 5.6 Effectiveness of maintaining two queues of prefetching sub-requests

In SoAP, we maintain two queues, namely A-Queue and I-Queue, to accommodate the active and inactive sub-requests, respectively. With time changing, degradation and eviction operations are performed based on the comparison between the currentT and the timestamps of sub-requests. Different queues support different scheduling policies. For example, the insertion and continuous scheduling policies are both available for A-Queue, while only the continuous scheduling policy is available for I-queue. This mechanism guarantees that the active sub-requests will be fetched in time and the inactive sub-requests will be performed when a cache miss hits them. In this experiment, we take a closer look at whether this double-queue mechanism is worthwhile.

In Fig. 13, SoAP is compared with a modified version of itself, where all the sub-requests including active and inactive ones are treated in the same way. In the modified one-queue SoAP, if a prefetch-

ing sub-request stays in the queue for a predefined time without any access, then it will be directly cancelled. We run this experiment with varied load under trace Fin2. As we can see, SoAP outperforms the one-queue SoAP by 11.3% in average response time when the load reaches Fin2-8. Even under Fin2-1, there is a performance gap of 8.6%. These results indicate that SoAP with one queue is more sensitive to the changes of load. That is, when the load becomes heavier, the one-queue SoAP still tries to perform the inactive sub-requests, leading to worse performance compared with SoAP. As a result, it is believed that the two-queue mechanism in SoAP is necessary and effective.

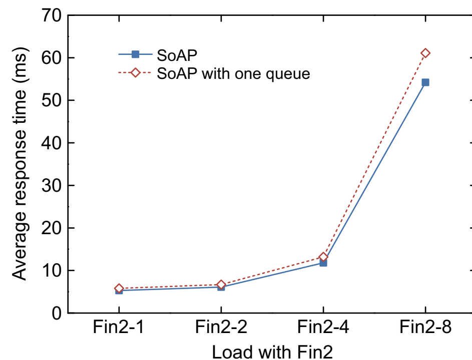


**Fig. 13 The performance test of maintaining two sub-request queues**

### 5.7 Idle bandwidth absorption

By asynchronous scheduling, SoAP absorbs the idle bandwidth of the disks for performing prefetching. By doing this, it reduces the bandwidth competition between user requests and prefetching sub-requests and lowers the prefetching cost. The benefit can be reflected by the average response time of user I/Os. For example, when the load bursts in a disk, the prefetching I/Os assigned to this disk will be performed later according to the time limit of its expectedT. As a result, the response time curve of user requests could be smoother. In Fig. 14, we evaluate the efficiency of idle bandwidth absorbing in SoAP. In this experiment, five disks are configured at the RAID-5 level with the strip unit size of 64 KB. Moreover, a 100 s trace is selected from Fin1, which is further divided into 50 equally sized intervals.

As shown in Fig. 14, although SEQP and SP improve the performance of response time, they cannot efficiently alleviate the load bursts. In contrast, SoAP not only improves the latency but also reduces time peaks during load bursts by asynchronous scheduling. Thus, it satisfies the needs of the users who are sensitive to request latency.

## 6 Related works

Sequential prefetching is the most popular prefetching scheme in modern data storage systems, and has been studied by many researchers. Most of these studies focus on achieving low prefetching cost, high prediction accuracy, and cache size economy.

### 6.1 Lower prefetching cost

For the storage system, the main cost of prefetching derives from the time it takes to bring the required disk blocks into memory. By identifying the sequential access pattern, sequential prefetching schemes commonly issue aggressive large-size prefetching requests to reduce the prefetching cost. For example, STEP (Liang *et al.*, 2007) is inclined to perform aggressive prefetching to hide disk access latency and reduce the number of expensive disk

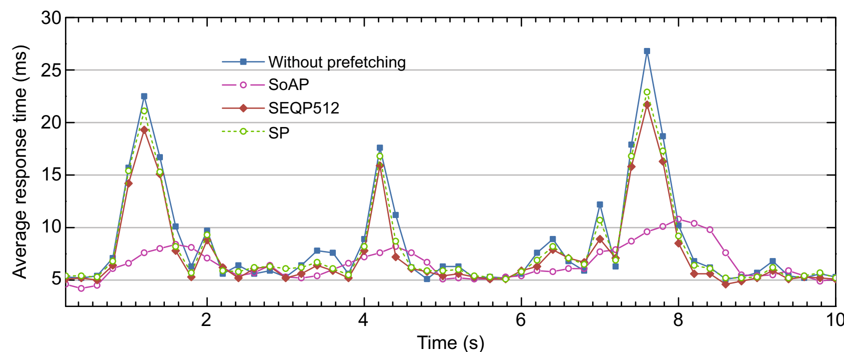


**Fig. 14 The average response time test with different prefetching strategies under Fin1**

operations. AMP (Gill *et al.*, 2007) gradually and heuristically adjusts the prefetching degree and trigger distance on a per-stream basis so as to achieve the highest possible aggregate throughput. Specifically, the size of the prefetching request is incremented by a fixed size whenever the last prefetching page is accessed. On the other hand, the prefetching size is reduced whenever the last prefetching pages reach the end of the LRU queue, which means that they will never be used. This algorithm always tries to maximize the prefetching degree until the cache capacity is insufficient. However, these schemes are suboptimal for the parallel disk system since they fail to consider the problems of sequentiality loss and synchronous fetching.

### 6.2 Improving prediction accuracy

Prediction accuracy is usually regarded as important due to the high penalty of miss prediction such as disk bandwidth waste, cache pollution, and pre-hit eviction of prefetched data (Liang *et al.*, 2007). Some studies proposed enhanced I/O interface to facilitate the prediction by getting more semantic information from applications. Cao *et al.* (1996) proposed to integrate application-controlled prefetching, caching, and scheduling for file systems. Chang and Gibson (1999) suggested leveraging the application hints by speculative execution without modifying the code. However, these schemes involve considerable computation overhead or reconstruction of applications.

History-based prediction, which allows an easier deployment, has been extensively studied. C-Miner (Li *et al.*, 2004) performs prefetching by mining the correlation between data blocks. TAP (Li *et al.*, 2008) adopts a table based prefetching approach, where only the addresses of the accessed blocks are stored. Therefore, there is more possibility for TAP to identify potential sequential streams in the workload by reserving more history information. STEP (Liang *et al.*, 2007) also uses a table to detect the sequential streams. This table is organized as a balanced tree and each node represents a detected or new sequential stream. Similarly, we use a hash table in SoAP to store the history information. Unlike them, we use the relationship graph and a cost-benefit model to assist our decision in prefetching so as to improve the prediction accuracy.

### 6.3 Prefetch cache management

Since cache is the scarce resource in the computer system, many studies focus on how to efficiently manage the prefetched data. The SARC algorithm (Gill and Modha, 2005) splits the cache into prefetch cache and re-reference cache. It focuses on balancing the cache allocation between them to adapt to the sequentiality of the workload. Li and Shen (2005) proposed a prefetch cache sizing scheme based on a gradient descent-based greedy algorithm. TAP (Li *et al.*, 2008) uses a downward pressure approach to reduce the prefetch size when the cache hit rate is stable. It also evicts the data of the previous request on a cache hit of the arriving request in the already detected sequential steam. The purpose is to keep the prefetch cache as small as possible while keeping a stable hit rate. SoAP does not implement any replacement algorithm for cache management, because it is not the focus of this work. Instead, it can flexibly accommodate the many cache replacement schemes proposed in other studies.

### 6.4 Prefetching for the parallel disk system

The most similar study to SoAP is ASP (Baek and Park, 2008), which is also dedicated to the parallel disk system. ASP observes that independence loss is the main cause of the poor performance of prefetching in the parallel storage system; thus, it proposes a strip prefetching approach to exploit the sequentiality in the single disk. It also improves cache utilization by balancing the prefetched pages and accessed pages, where it tends to cull the useless prefetched data earlier. However, ASP handles cache miss in an over simple way, where the complete strip will be read when a cache misses hitting any of its blocks. As a result, ASP has the problem of low prediction accuracy. Additionally, it is unable to solve the problem of synchronous fetching. Unlike ASP, SoAP performs prefetching by sequential stream detection with high prediction accuracy and asynchronously schedules the prefetching subrequests.

## 7 Conclusions

The parallel disk system, which plays a central role in the storage sub-system, has its own distinct physical data layout, where logical contiguous blocks

are physically sequential only if they are in the same strip. This special feature introduces both challenges and opportunities for the design of sequential prefetching techniques. Considering the technology trend that storage servers are equipped with more processors and increased memory capacity, it is worthwhile to incorporate more sophisticated and powerful sequential prefetching schemes to improve the performance of sequential accesses.

In this study, we have introduced the relationship graph as a tool to determine the hit probability of a prefetching request in a stream with a given length. Moreover, we have incorporated this tool into our cost-benefit model in order to explore the optimal prefetching length according to the system load status and sequentiality degree, which avoids the performance penalty and cache pollution caused by inaccurate prefetching.

We have designed a strip-oriented prefetching scheme to adapt the prefetching operation to the physical data layout of the parallel disk system, by which every prefetching request must be aligned in the strip and then split into multiple sub-requests. Thus, the basic granularity is based on the strip size, which allows exploiting the maximum available physical sequentiality.

We have also implemented an asynchronous scheduling policy along with two queues to manage the prefetching sub-requests of each disk. Based on the timestamps of each sub-request and the observation of disks' load status, scheduling policies are performed to load the desired data timely and economically.

We believe that the insight of SoAP is widely applicable not only in parallel disk systems, but in any system that has striping mechanism and sequential workload.

## References


Baek, S.H., Park, K.H., 2008. Prefetching with Adaptive Cache Culling for Striped Disk Arrays. USENIX ATC, p.363-376.

Bhatia, S., Varki, E., Merchant, A., 2010. Sequential Prefetch Cache Sizing for Maximal Hit Rate. MASCOTS, p.89-98. [doi:10.1109/MASCOTS.2010.18]

Bovet, D., Cesati, M., Oram, A., 2005. Understanding the Linux Kernel. O'Reilly, Sebastopol, CA, USA.

Bowman, I.T., Salem, K., 2005. Optimization of query streams using semantic prefetching. *ACM Trans. Database Syst.*, **30**(4):1056-1101. [doi:10.1145/1114244.1114250]

Cao, P., Felten, E.W., Karlin, A.R., Li, K., 1996. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, **14**(4):311-343. [doi:10.1145/235543.235544]

Chang, F., Gibson, G.A., 1999. Automatic I/O Hint Generation Through Speculative Execution. OSDI, p.1-14.

Gill, B.S., Modha, D.S., 2005. SARC: Sequential Prefetching in Adaptive Replacement Cache. USENIX ATC, p.293-308.

Gill, B.S., Angel, L., Bathen, D., 2007. AMP: Adaptive Multi-stream Prefetching in a Shared Cache. FAST, p.185-198.

Hartung, M., 2003. IBM total storage enterprise storage server: a designer's view. *IBM Syst. J.*, **42**(2):383-396. [doi:10.1147/sj.422.0383]

Hsu, W.W., Smith, A.J., Young, H.C., 2001. I/O reference behavior of production database workloads and the TPC benchmarks—an analysis at the logical level. *ACM Trans. Database Syst.*, **26**(1):96-143. [doi:10.1145/383734.383737]

Kamruzzaman, M., Swanson, S., Tullsen, D.M., 2011. Inter-Core Prefetching for Multicore Processors Using Migrating Helper Threads. ASPLOS, p.393-404. [doi:10.1145/1950365.1950411]

Li, C., Shen, K., 2005. Managing Prefetch Memory for Data-Intensive Online Servers. FAST, p.253-266.

Li, M.J., Varki, E., Bhatia, S., Merchant, A., 2008. TAP: Table-Based Prefetching for Storage Caches. FAST, p.1-16.

Li, Z.M., Chen, Z.F., Srinivasan, S.M., Zhou, Y.Y., 2004. C-Miner: Mining Block Correlations in Storage Systems. FAST, p.173-186.

Liang, S., Jiang, S., Zhang, X.D., 2007. STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers. ICDCS, p.64-73. [doi:10.1109/ICDCS.2007.141]

Lymberopoulos, D., Riva, O., Strauss, K., Mittal, A., Ntoulas, A., 2012. Pocketweb: Instant Web Browsing for Mobile Devices. ASPLOS, p.1-12. [doi:10.1145/2150976.2150978]

RAID Advisory Board, 1999. The Raidbook: a Source Book for RAID Technology (6th Ed.). Lino Lakes, MN.

Storage Performance Council, 2011. SPC Benchmark 2/Energy (SPC-2/E), SPC Benchmark 2C/Energy (SPC-2C/E) Benchmark Extensions Address Energy Use in Sequential Applications. Available from http://www.storageperformance.org/press/SPC_2E_2CE_PR_final.pdf [Accessed on Jan. 13, 2012].

Tian, L., Feng, D., Jiang, H., Zhou, K., Zeng, L.F., Chen, J.X., Wang, Z.K., Song, Z.L., 2007. PRO: a Popularity-Based Multi-threaded Reconstruction Optimization for RAID-Structured Storage Systems. FAST, p.277-290.

Zhang, Z., Kulkarni, A., Ma, X.S., Zhou, Y.Y., 2009. Memory Resource Allocation for File System Prefetching: from a Supply Chain Management Perspective. EuroSys, p.75-88. [doi:10.1145/1519065.1519075]