Prefetching with Adaptive Cache Culling for Striped Disk Arrays

Sung Hoon Baek and Kyu Ho Park
Korea Advanced Institute of Science and Technology
shbaek@core.kaist.ac.kr,kpark@ee.kaist.ac.kr

Abstract

Conventional prefetching schemes regard prediction accuracy as important because useless data prefetched by a faulty prediction may pollute the cache. If prefetching requires considerably low read cost but the prediction is not accurate, it may or may not be beneficial depending on the situation. However, the problem of low prediction accuracy can be dramatically reduced if we efficiently manage prefetched data by considering the total hit rate for both prefetched data and cached data. To achieve this goal, we propose an adaptive strip prefetching (ASP) scheme, which provides low prefetching cost and evicts prefetched data at the proper time by using differential feedback that maximizes the hit rate of both prefetched data and cached data in a given cache management scheme. Additionally, ASP controls prefetching by using an online disk simulation that investigates whether prefetching is beneficial for the current workloads and stops prefetching if it is not. Finally, ASP provides methods that resolve both independency loss and parallelism loss that may arise in striped disk arrays. We implemented a kernel module in Linux version 2.6.18 as a RAID-5 driver with our scheme, which significantly outperforms the sequential prefetching of Linux from several times to an order of magnitude in a variety of realistic workloads.

1 Introduction

Prefetching is necessary to reduce or hide the latency between a processor and a main memory as well as between a main memory and a storage subsystem that consists of disks. Some prefetching schemes for processors can be applied to prefetching for disks by means of a slight modification. And many prefetching techniques that are dedicated to disks have been studied. We focus on disk prefetching, especially for striped disk arrays.

The frequently-addressed goal of disk prefetching is to make data available in a cache before the data is consumed; in this way, computational operations are overlapped with the transfer of data from the disk. The other goal is to enhance the disk throughput by aggregating multiple contiguous blocks as a single request. Prefetching schemes for a single disk cause problems is used in striped disk arrays. There is a need for a special scheme for multiple disks, in which the characteristics of striped disk arrays [7] are considered.

1.1 Five Problems

The performance disparity between processor speed and the disk transfer rate can be compensated for via the disk parallelism of disk arrays. Chen et al. [7] described six types of disk arrays and termed them redundant disk arrays of independent disks (RAID). In these arrays, blocks are striped across the disks, and the striped blocks provide the parallelism of multiple disks, thereby improving access bandwidth. Many RAID technologies have focused on the following: reliability of RAID [1, 5, 16, 34, 40], the write performance [3, 12, 46], multimedia streaming with a disk array [15, 21, 27], RAID management [45, 47], and so on [20, 41].

However, prefetching schemes for disk arrays have been rarely studied. Some offline prefetching schemes (described in Section 1.2.3) take load balancing among disks into account, though they are not practical since they require complete knowledge of future I/O references.

For multiple concurrent reads, the striping scheme of RAID resolves the load balancing among disks [7]. The greater number of concurrent reads implies more evenly distributed reads across striped blocks. As a result, researchers are forced to address five new problems that substantially affect the prefetching performance of striped disk arrays as follows:

1.1.1 Parallelism

If the prefetch size is much less than the stripe size and the number of concurrent reads is much less than the number of the member disks that compose a striped disk array, some of disks become idle, thereby losing parallelism of the disks. This case exemplifies what we call *parallelism loss*.

In sequential prefetching (described in Section 1.2.4), a large prefetch size laid across multiple disks can prevent parallelism loss. For the worst case of a single stream, the prefetch size must be equal to or larger than the stripe size. However, this method gives rise to prefetching wastage [10] for multiple concurrent sequential reads, which are common in streaming servers. A streaming server that serves tens of thousands of clients,

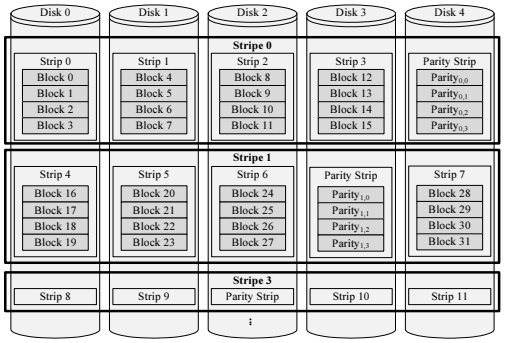


Figure 1: The data organization and terminologies of a RAID-5 array

requires several gigabytes of prefetching memory. If the server does not have an over-provisioned memory, thrashing occurs as the prefetched pages are replaced too early by massive prefetches before they are used.

1.1.2 Independency

Traditional prefetching schemes have focused on parallelism or the load balance of disks. However, this paper reveals that the *independency* of disks is more important than *parallelism* for concurrent reads of large numbers of processes in striped disk arrays, whereas parallelism is only significant when the number of concurrent accesses is roughly less than the number of member disks.

A strip is defined by the RAID Advisory Board [39] as shown in Fig. 1, which illustrates a RAID-5 array consisting of five disks. The stripe is divided by the strips. Each strip is comprised of a set of blocks.

Figure 2(a) shows an example of *independency loss*. The prefetching requests of conventional prefetching schemes are not aligned in the strip; therefore, a single prefetching request may be split across several disks. In Fig. 2(a), three processes request sequential blocks for their own files. A preset amount of sequential blocks are aggregated as a single prefetching request by the prefetcher. If each prefetch request is not dedicated to a single strip, it is split across two disks, and each disk requires two accesses. For example, if a single prefetch request is for Block 2 to Block 5, the single prefetch request generates two disk commands that correspond to Block 2 & 3 belonging to Disk 0 and Block 4 & 5 belonging to Disk 1. This problem is called *independency loss*. In contrast, if each prefetching request is dedicated to only one disk, as shown in Fig. 2(b), independency loss is resolved.

To resolve independency loss, each prefetching request must be dedicated to a single strip and not split to multiple disks. The strip size is much less than the stripe size and, as a result, suffers parallelism loss for small numbers of concurrent reads. The two problems,

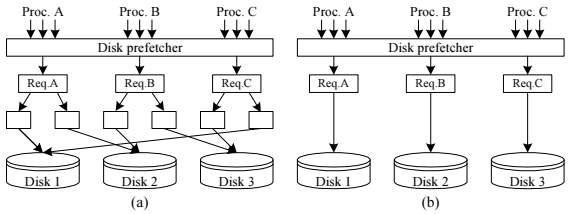


Figure 2: (a) Independency loss: prefetch requests are split across multiple disks so that each prefetch requires two disk accesses. (b) No independency loss: each prefetch request is dedicated to one disk.

independency loss and parallelism loss, conflict with one other. In the traditional prefetching schemes, if one problem is resolved, the other arises.

We propose an adaptive strip prefetching (ASP) scheme that eliminates almost all demerits of a new strip prefetching scheme, as discussed in Section 2.1. The proposed strip prefetching prefetches all blocks of a strip of a striped disk array on a cache miss. The strip is a set of contiguous blocks dedicated to only one disk, ASP including strip prefetching resolves independency loss. Parallelism loss is resolved by combining ASP with our earlier work, massive stripe prefetching (MSP) [2], which maximizes parallelism for sequential reads of a small number of processes.

1.1.3 Non-Sequential Read and Small Files

Sequential prefetching is a widely used practical prefetching scheme. However, it is never beneficial to non-sequential reads, although they may exhibit some spatial locality. In addition, the prefetch size of sequential prefetching is restricted by file size. Although a single process sequentially reads contiguous small files, the small prefetch size of sequential prefetching causes parallelism loss. Such workloads are common in real systems but there is no scheme that has the advantages of being; (1) beneficial to them, (2) practical for real systems, (3) of low overhead, (4) transparent to applications, and (5) convenient to use.

ASP improves the performance of such workloads because strip prefetching exploits the principle of the spatial locality by implicitly prefetching data that is likely to be referenced in the near future.

1.1.4 Prefetched Data Management

Traditional prefetching schemes regard prediction accuracy as important because there is no efficient and practical cache management scheme for uselessly prefetched data that pollute the cache. Strip prefetching requires considerably low read cost but its prefetching prediction is not so accurate. Consequently, the overall throughput of strip prefetching may or may not be beneficial de-

pending on the situation. However, the problem of low prediction accuracy can be dramatically reduced if we efficiently manage prefetched data by considering the total hit rate for both prefetched data and cached data.

We need to divide the traditional meaning of cache hit into prefetch hit and the narrow interpretation of cache hit. The **prefetch hit** is a read request on a prefetched data that has not yet been referenced by the host since it was prefetched. The **cache hit** is defined as a read request on a cached data that was loaded by the host's past request.

With the absence of efficient cache management for prefetched data, the low prediction accuracy of strip prefetching exhausts the cache memory. To resolve this problem, ASP early evicts (culls) prefetched data that is less valuable than the least-valuable cached data in an adaptive manner with differential feedback so that *the sum of the cache hit rate and the prefetch hit rate of ASP is maximized in a given cache management scheme*. The differential feedback has similar features with the adaptive scheme based on marginal utility used in the sequential prefetching in adaptive replacement cache (SARC) [11]. However, there are several differences between SARC and our scheme, which are discussed at the end of Section 2.3.

Several approaches for balancing the amount of memory given to cached data and prefetched data were taken by Patterson et al's transparent informed prefetching [35] and its extension [42]. However, there are many significant differences between them and our scheme. Section 1.2.5 addresses these in more detail.

ASP does not decide which cached data should be evicted and most cache replacement schemes do not take the prefetch hit into account. Hence, ASP may perform well with any of the recent cache replacement policies including RACE [48], ARC [29], SARC [11], AMP [10], and PROMOTE [9].

1.1.5 Prefetching Cost

As well as requiring prefetched data management, prefetching must spend less time in reading the requested data from disks. Hence, ASP include a simple online disk simulation to estimate whether strip prefetching requires a larger read cost than no prefetching for the current workload. ASP activates or deactivates strip prefetching depending on this comparison.

If a workload exhibits neither prefetch hits nor cache hits, it is apparent that strip prefetching has a greater cost than no prefetching. In this case, any prefetching should be deactivated, even though ASP efficiently culls uselessly prefetched data. For real workloads, ASP exploits an online disk simulation to estimate the two read costs.

1.2 Related Work

1.2.1 History-Based Prefetching

History-based prefetching, which predicts future accesses by learning the stationary past accesses, has been proposed in various forms. Palmer and Zdonik proposed an associated memory that recognizes access patterns by repeated training [33]. Grimsrud et al. provided an adaptive table, in which an entry is associated with each cluster (of one or more disk blocks) on the disk; furthermore, each entry contains the next cluster for the best prefetching candidate and weight [14]. Griffioen and Appleton suggested a file-level prediction that prefetches files early based on the past file activity [13]. Prefetching with a Markov predictor, which is based on the transition frequency of reference strings, has also been studied [18]. Recording and analyzing past accesses requires a significant amount of memory; hence, several studies have been proposed to reduce the required amount of memory for their history-based prefetching [23, 25, 44].

History-based prefetching, which records and mines the extensive history of past accesses, is cumbersome and expensive to maintain in practical systems. It also suffers from low predictive accuracy, and the resultant unnecessary reads can degrade performance. Furthermore, it is effective only for stationary workloads.

1.2.2 Application-Hint-Based Prefetching

When a small number of processes or a single process generates a non-sequential access, a small number of concurrent I/Os may not fully exploit disk parallelism in the disk array. In order to solve this problem, Patterson et al. suggested a disclosure hint interface [35], This interface must be exploited by an application programmer so that information about future accesses can be given through an I/O-control (ioctl) system call. The state-of-art interface, the asynchronous I/O of Linux 2.6 [4], may replace the ioctl system call. The disclosure hint forces programmers to modify applications so that the applications issue hints. Some applications involve significant code restructuring to include disclosure hints.

Speculative execution provides application hints without modifying the code [6]. A copy of the original thread is speculatively executed without affecting the original execution in order to predict future accesses. However, the speculative thread consumes considerable computational resources and can cause misprediction if the future accesses rely on the data of past accesses.

1.2.3 Offline Optimal Prefetching

Traditional buffer management algorithms that minimize cache misses are substantially suboptimal in parallel I/O systems where multiple I/Os can proceed simultaneously [19]. Analytically optimal prefetching and caching

schemes have been studied with respect to situations in which future accesses are given [19, 22]. These schemes are optimal in terms of cache hit rate and disk parallelism. As a metric value, however, the cache hit rate may not accurately reflect the real performance because a sequential read for tens of blocks can achieve a much higher disk throughput than random reads for two blocks [8]. Furthermore, offline prefetching does not resolve the two conflicting problems of parallelism loss and independency loss.

1.2.4 Sequential Prefetching

The most common form of prefetching is sequential prefetching (SEQP), which is widely used in a variety of operating systems because sequential accesses are common in practical systems.

The *Atropos* volume manager [36] dramatically reduces disk positioning time for sequential accesses to two dimensional data structures by means of new data placement in disk arrays. However, *Atropos* does not give no solution at all to the five problems addressed in Introduction.

Table-based prefetching (TaP) [26] detects these sequential patterns in a storage cache without any help of file systems, and dynamically adjusts the cache size for sequentially prefetched data, namely prefetch cache size, which is adjusted to an efficient size that obtains no more prefetch hit rate above the preset level even if the prefetch cache size is increased. However, TaP fails to consider both prefetch hit rate and cache hit rate.

MSP, which was proposed in our earlier work [2], detects semi-sequential reads at the block level. A sequential read at the file level exhibits a *semi-sequential* read at the block level because reads at the block level require both metadata access and data access throughout in different regions and the file may be fragmented. If a long semi-sequentiality is detected, the prefetch size of MSP becomes a multiple of the stripe size, and the prefetching request of MSP is aligned in the stripe; as a result, MSP achieves perfect disk parallelism.

Although sequential prefetching is the most popular prefetching scheme in practical systems, sequential prefetching and its variations have until now failed to consider independency loss and they are not at all beneficial to non-sequential reads.

1.2.5 Prefetching and Caching

Among offline prefetching schemes, Kallahalla [19] and Kimbrel [22] took both prefetched data and cached data into account in order to increase cache hit rate. However, their approaches are not realizable in actual systems since they require complete knowledge of future accesses. Patterson et al. [35, 42] provided practical schemes known as TIP and TIPTOE (TIP with Temporal Overload Estimators, an extension of TIP), TIP and TIPTOE estimate the read service time of prefetched data, the hinted cache, and the shrinkage of the cached data. They then choose the globally least-costly block cache for the victim of eviction.

Distinguishing prefetched data from cached data is the common part of TIP, TIPTOE, and our ASP. However, there are notable differences between the first two and ASP. (1) While their eviction policy does not manage uselessly prefetched data, ASP can manage inaccurately prefetched data that are prestaged by strip prefetching. This provides noticeable benefits if the workload exhibits spatial locality. (2) They are based on an approximate I/O service time model to assess the costs for each prefetched block. This deterministic cost estimation may be different from the real costs and cause errors, while ASP uses an adaptive manner that measures and utilizes the instantaneous real cost of prefetched data and cached data. (3) They must scan block caches to find the least-valuable one, while ASP has a negligible overhead with O(1) complexity.

2 Adaptive Strip Prefetching

To resolve the five problems mentioned in Introduction, we propose adaptive strip prefetching (ASP), which includes three new schemes, *strip prefetching*, *adaptive cache culling*, and an *online disk simulation*.

Strip prefetching resolves independency loss by dedicating each prefetching request to a single disk, and resolves parallelism loss when combined with MSP [2]. Adaptive cache culling eliminates the inaccurate prediction of strip prefetching by providing an optimal point that improves both cache hit rate and prefetch hit rate. To guarantee no performance degradation, ASP enables or disables strip prefetching by using a simple online disk simulation. The following sections describe the three new components comprising ASP.

2.1 Strip Prefetching

To resolve independency loss and to be beneficial for non-sequential reads as well as sequential reads, we propose strip prefetching. Whenever the block requested by the host is not in the cache, strip prefetching reads all blocks of the strip to which the requested block belongs. By grouping consecutive blocks of each strip into a single prefetch unit, strip caches exploit the principle of spatial locality by implicitly prefetching data that is likely to be referenced in the near future. Because each strip is dedicated to only one disk, each prefetch request is not laid across multiple disks; as a result, the problem of independency loss is resolved.

However, strip prefetching has two drawbacks. First, strip prefetching may degrade memory utilization by prefetching useless data. Hence, we propose adaptive

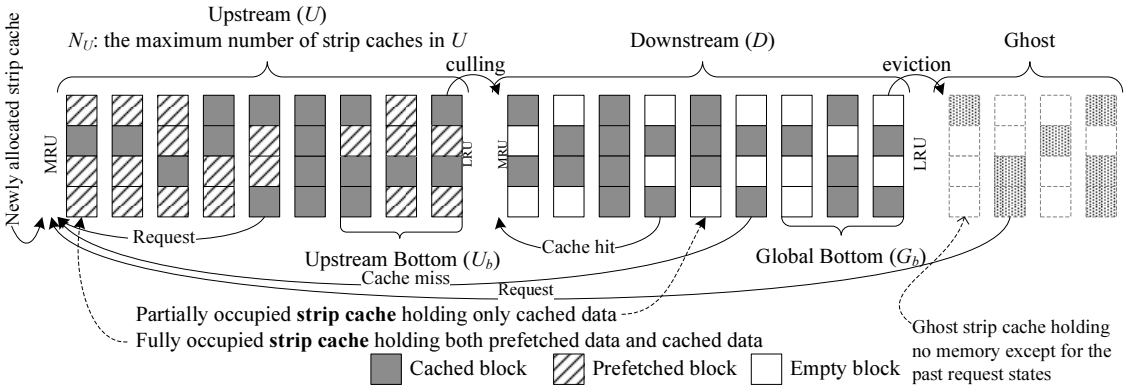


Figure 3: The cache management of ACC: the cache is partitioned into the upstream U and the downstream D. The upstream bottom U_b is a bottom portion of U. The global bottom G_b is a bottom portion of the global list consisting of U and D. ACC changes the variable N_U, the maximum number of strip caches that U can accommodate, in an adaptive manner. If the number of strip caches in U exceeds N_U, ACC evicts (culls) the prefetched block caches of the LRU strip cache of U and moves the strip cache into the MRU position of D.

cache culling to improve strip prefetching. Adaptive cache culling evicts (culls) prefetched and unused data at the proper time in an adaptive manner; as a result, the sum of the cache hit rate and the prefetch hit rate of ASP is guaranteed to be equal to or larger than those of either strip prefetching or no prefetching.

Second, the read service time of strip prefetching may be longer than that of no prefetching. Hence, ASP activates or deactivates strip prefetching by estimating whether the read cost of is less than not prefetching for the current workload. The estimation can be performed by an online disk simulation with low overhead.

2.2 Adaptive Cache Culling

To efficiently manage data prefetched by strip prefetching, as a component of ASP, we propose an adaptive cache culling (ACC) scheme, which maximizes the sum of the cache hit rate and the prefetch hit rate (see Section 1.1.4 for this terminology) of ASP in a given cache management scheme.

Figure 3 illustrates the cache structure that is managed in strip caches, each of which consists of four blocks. ACC manages the cache in strip units. Each strip cache holds the data blocks of a strip. The data block can be a cached block holding cached data, a prefetched block holding prefetched data, or an empty block holding neither memory nor data.

To evict prefetched data earlier than cached data at the proper time in an adaptive manner (described in Section 2.3), as shown in Fig. 3, strip caches are partitioned into the upstream U and the downstream D. U can include both prefetched blocks and cached blocks but D excludes prefetched blocks. Newly allocated strip cache that may hold both prefetched blocks and cached blocks is inserted into U. If the number of strip caches in U exceeds the maximum number of strip caches that U can accommodate, the LRU strip cache of U moves to D, and ACC **culls** (evicts) the prefetched blocks of this strip cache.

A host request that is delivered to a prefetched block changes it into a cached block. All cached or prefetched blocks make their way from upstream toward downstream like a stream of water. If a prefetched block flows downstream, the prefetched block is changed into an empty block by culling, which deallocates the memory holding the prefetched data but retains the information that the block is empty in D. Thus, blocks requested by the host remain in the cache for a longer time than prefetched and unrequested blocks. The average lifetime of prefetched blocks is determined by the size of U, which is dynamically adjusted by measuring instant hit rates.

ACC changes a variable N_U, the maximum number of strip caches that U can accommodate, in an adaptive manner. If N_U decreases, the prefetch hit rate decreases due to a reduced amount of prefetched data but the cache hit rate increases, otherwise, vice versa. The system performance depends on the total hit rate that is the sum of the prefetched hit rate and the cache hit rate. Section 2.3 describes a control scheme for N_U to maximize the total hit rate, while this section describes the cache management and structure of ACC.

The LRU strip cache of all is evicted under memory pressure. A cache hit for a strip cache of U and D moves the corresponding strip cache to the MRU position of U and D, respectively, like the least recently used (LRU) policy. An evicted strip cache can be a ghost strip cache, which is designed to improve the performance using request history. The ghost strip cache has a loose relationship with the major culling process. Hence, ACC can perform even if we do not manage the ghost strip cache.

The next section describes the ghost strip cache and some issues.

2.2.1 Issues

The stream management of ACC prevents the cache hits in D from moving the hit strip cache to U. If a partially occupied strip cache (including an empty block) of D can move to U, the memory space occupied by U is shrunk because the partially occupied strip cache may evict another fully occupied strip cache from U to keep the number of strip caches of U from being equal to N_U. This process breaks the optimal partition of U and D.

Sibling strip caches are the strip caches corresponding to the strips that belong to the same stripe. Destaging dirty blocks from the RAID cache is more efficient when it is managed in stripes than in blocks or strips [12]. The stripe cache contains sibling strip caches belonging to the stripe. Hence, although a strip cache is evicted, its stripe cache is still alive if at least one sibling strip cache is alive. Therefore, the stripe cache can easily maintain an evicted strip cache as a ghost strip cache.

When all blocks of a strip cache are evicted from the cache, the strip cache becomes a *ghost strip cache* if one of its sibling strip caches is still alive in the cache. Ghost strip caches hold no memory except for request states of the past. The request states indicate which blocks of the strip cache were requested by the host. When an I/O request is delivered for a ghost strip, the ghost strip becomes alive as a strip cache that retains the past request states. This strip cache that was a ghost is called a reincarnated strip cache, but which has no distinction with the other strip caches except for the past request states.

The past request states of the ghost strip cache works like a kind of history-based prefetching. The past request states remain even after the ghost strip cache is reincarnated. The past cached block, which had not been requested before the block became a ghost, has high possibility of being referenced by the host in the near future although it has not yet been referenced since it was reincarnated. Thus, *the culling process does not evict past cached blocks as well as cached blocks*.

The cache replacement policy in our implementation is to evict the LRU strip cache of all. However, ACC does not determine a cache replacement policy that may evict any cached block in U or D. A new policy for ACC or a combination of a state-of-the-art cache replacement policy as referenced in Section 1.1.4 and ACC may provide better performance than the LRU scheme. The exact cache replacement policy is not within the scope of this paper.

2.2.2 Summary of Operation

Whenever a request from the host is delivered to the disk array, the two lists, U and D, are managed by the following rules:

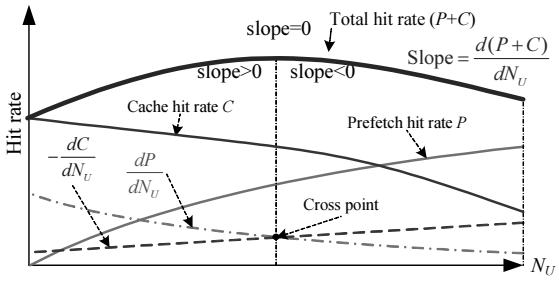


Figure 4: The function of total hit rate ($P + C$) with respect to N_U: when the slope of the function is zero, the total hit rate is at the maximum.

- *A cache hit or prefetch hit occurring in U:* The strip cache that corresponds to the hit moves to the MRU position of U.
- *A cache hit occurring in D:* The strip cache moves to the MRU position of D.
- *A request for a ghost strip cache or an empty block of alive strip caches:* The corresponding strip is read from the disk by controlled strip prefetching (as described in Section 2.4, strip prefetching can be deactivated by the decision algorithm of ASP with an online disk simulation.), and its strip cache is inserted into the MRU position of U.
- *A cache miss on neither alive nor ghost strip caches:* A new strip cache is allocated for the requested block, read by the controlled strip prefetching, and inserted into the MRU position of U.
- *If the number of strip caches exceeds N_U,* the LRU strip cache of U moves to the MRU position of D, and the prefetched blocks except for the past cached blocks are culled (evicted).

2.3 Differential Feedback of ACC

Finding the optimal value of N_U, the maximum number of strip caches occupying U, is the most important part of ACC. We regard N_U as optimal when it maximizes the total hit rate, which is the sum of the prefetch hit rate P and the cache hit rate C. Fig. 4 illustrates a function of the total hit rate with respect to N_U. When the slope of the function is zero, the total hit rate is at the maximum.

As N_U increases, P increases but the incremental rate of P declines. As N_U decreases, C increases and the incremental rate of C declines. Therefore, the derivative of P with respect to N_U, dP/dN_U, is a *monotonically decreasing function* and so is the negative derivative of C with respect to N_U, $-dC/dN_U$. Then, there is **zero or one cross point** of these derivatives. The function of $P + C$ with respect to N_U can form a hill shape or be an monotonically increasing or deceasing function, these three types of functions have only one peak point.

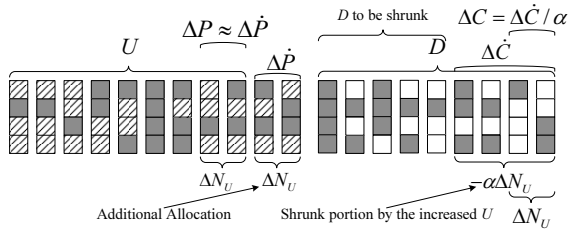


Figure 5: The derivative of P with respect to N_U is similarly equal to the increased number of prefetch hits in U_b when N_U increases slightly. The derivative of C is negatively proportional to the decreased number of cache hits in G_b when the size of U increases.

If we can determine the instant value of the slope for the current N_U, the optimal N_U can be achieved using the following feedback with the instant value of the slope.

$$N_U \leftarrow N_U + C \times \text{slope}, \tag{1}$$

where C is a constant value that determines the speed of the feedback.

When N_U is on the left side of Fig 4, the feedback increases N_U by the positive value of the slope. If N_U increases so much that the slope becomes negative, the feedback with the negative slope decreases N_U so that the total hit rate is near the peak. Even if $P + C$ is an increasing or decreasing function, the above feedback finds the optimal value of N_U.

The function of the slope is the derivative of the total hit rate $P + C$, which is the sum of the two derivatives of P and C with respect to N_U as the following equation.

$$\text{slope} = \frac{d(P+C)}{dN_U} = \frac{dP}{dN_U} + \frac{dC}{dN_U}. \tag{2}$$

An approximated derivative can be measured as shown in Fig. 5. By the definition of differential, the derivative of P is similarly equal to the number of prefetch hits $\Delta \dot{P}$ that, during a time interval, occur in the ΔN_U strip caches additionally allocated to U. The additional prefetch hit rate $\Delta \dot{P}$ is similar to ΔP, which is the prefetch hit rate of the strip caches of the upstream bottom that is adjacent to the additionally allocated strip caches. Therefore, the derivative of the prefetch hit rate with respect to N_U can be approximated with the following equation.

$$\frac{dP}{dN_U} \simeq \frac{\Delta \dot{P}}{\Delta N_U} \simeq \frac{\Delta P}{\Delta N_U}. \tag{3}$$

If the upstream U increases by ΔN_U, the downstream D decreases by $\alpha \Delta N_U$. The coefficient α is determined by the occupancy ratio of the non-empty blocks in both the expanded portion of U and the shrunk portion of D (see Eq. (6)). The derivative of the cache hit rate with respect to N_U is similar to the cache hit rate $\Delta \dot{C}$ that occurs in the $\alpha \Delta N_U$ strip caches of the global bottom. If we want to monitor the cache hit rate, ΔC, in a fixed size of the global bottom, the derivative of the cache hit rate can be approximated with the following equation.

$$\frac{dC}{dN_U} \simeq -\frac{\Delta \dot{C}}{\Delta N_U} \simeq -\frac{\alpha \Delta C}{\Delta N_U}. \tag{4}$$

The differential value (slope) of the current partition can be obtained by monitoring the number of prefetch hits ΔP occurring in the upstream bottom and the number of cache hits ΔC occurring in the global bottom during a time interval.

$$\text{slope} \propto \Delta P - \alpha \Delta C. \tag{5}$$

The upstream bottom U_b is the bottom portion of U, where the number of strip caches of U_b is predetermined as 20% of the maximum number of strip caches of the simple strip prefetching policy. Similarly, the global bottom G_b is the bottom portion of the global list (the combination of U and D). At an initial stage or in some other cases, D contains no strip cache or too small number of strip caches. Hence, G_b can overlap with U_b. G_b accommodates the same number of strip caches as U_b.

By combining Eqs. (1) and (5), the final feedback operation that achieves the maximum hit rate can be written as in the following process.

$$\alpha = \frac{\text{the number of all blocks in } U_b}{\text{the number of cached blocks in } G_b}. \tag{6}$$

$$N_U \leftarrow N_U + S(\Delta P - \alpha \Delta C). \tag{7}$$

Whenever a hit occurs in U_b or G_b, ACC adjusts N_U using this process. The constant S of Eq. (7) determines the speed of the adaptation. In our experiment, S was speculatively chosen as two. We select the time interval to measure ΔP and ΔC to be the time difference between two successive hits occurring in either U_b or G_b. ΔP and ΔC are hence zero or one.

The upstream size N_U is initially set to the cache size over the strip size. Until the cache is fully filled with data, ASP activates strip prefetching and N_U is not changed by the feedback. At the initial time, D does not exist, and thus, G_b is identical to U_b.

We do not let N_U fall below the size of U_b to retain the fixed size of U_b. If N_U shrinks to the size of U_b, ACC deactivates strip prefetching until N_U swells to twice the size of U_b to make a hysteresis curve. The reason of this deactivation is that too small N_U indicates strip prefetching is not beneficial in terms of not only hit rate but also read service time.

The differential feedback has similar features with the adaptive manner based on marginal utility used in the sequential prefetching in adaptive replacement cache

(SARC) [11]. However, we present a formal method that finds an optimal point using the derivatives of cache hit rate and prefetch hit rate. Only our analytical approach can explain the feedback coefficient α. Furthermore, there are several differences between the two algorithms: (1) while SARC does not distinguish prefetched blocks from cached blocks, our scheme takes care of the eviction of only prefetched blocks that are ignored in SARC, (2) SARC relates to random data and sequential data, whereas the proposed scheme considers both prefetch hits and cache hits in a prefetching scenarios, and (3) our scheme manages the caches in terms of strips for an efficient management of striped disk arrays, while SARC has no feature for disk arrays.

2.4 The Online Cost Estimation

For example, if N_U decreases to the minimum value due to the lack of prefetch hits, ASP must deactivate strip prefetching. The deactivation, however, cannot rely on the minimum size of N_U because N_U may not shrink if both the prefetch hit rate and cache hit rate are zero or extremely low. Therefore, ASP employs an online disk simulation, which investigates whether strip prefetching requires a greater read cost than no prefetching.

Whenever the host requests a block, ASP calculates two read costs, C_n and C_{sp}, by using an online disk simulation with a very low overhead. The disk cost C_{sp} is the virtual read time spent in disks for all the blocks of the cache by pretending that strip prefetching has always been performed. Similarly, the disk cost C_n is the virtual read time spent in disks for all blocks of the current cache by pretending that no prefetching has ever been performed.

Whenever the host requests a block that is not in the cache, ASP compares C_n with C_{sp}. If C_n is less than C_{sp}, ASP deactivates strip prefetching for this request because strip prefetching is estimated to have provided slower read service time than no prefetching for all recent I/Os that have affected the current cache.

If a block hits the cache as a result of strip prefetching, the block must be one of all the blocks in the cache. The gain of strip prefetching is therefore correlated with all blocks in the cache that contains the past data. Hence, all blocks in the cache are exploited to determine whether strip prefetching provides a performance gain.

The cache is managed in terms of strips and consists of N_s strip caches, and each strip cache, S_i, has two variables, $\mathrm{c}_n(S_i)$ and $\mathrm{c}_{sp}(S_i)$, which are updated by the online disk simulation for every request from the host to the strip cache S_i. The variable $\mathrm{c}_n(S_i)$ is the virtual read time spent in S_i if we assume that no prefetching has ever been performed. The variable $\mathrm{c}_{sp}(S_i)$ is the virtual read time spent in S_i if we assume that strip prefetching has always been performed. Then we can express C_n and C_{sp} as follows:

$$\mathrm{C}_n \equiv \sum_{i=1}^{N_s} \mathrm{c}_n(S_i), \quad \mathrm{C}_{sp} \equiv \sum_{i=1}^{N_s} \mathrm{c}_{sp}(S_i), \tag{8}$$

where S_i is an i-th strip cache and $\{S_i | 1 \leq i \leq N_s\}$ is the entire cache that consists of N_s strip caches.

From Eq.(8), C_{sp} can be obtained by the sum of $\mathrm{c}_{sp}(S_i)$ for all strip caches. It seems to require a great overhead. However, an equivalent operation with O(1) complexity can be achieved as follows: (1) Whenever the host causes a cache miss on a new block whose strip cache is not in U, calculate the disk cost to read all blocks of the requested strip and add it to both $\mathrm{c}_{sp}(S_i)$ and C_{sp}; (2) whenever a read request accesses to a prefetched block or misses the cache, calculate the virtual disk cost to access to the block even though the request hits the cache, and add the disk cost to both C_n and $\mathrm{c}_n(S_i)$ of the strip cache S_i to which the requested block belongs; and (3) subtract $\mathrm{c}_{sp}(S_i)$ from C_{sp} and subtract $\mathrm{c}_n(S_i)$ from C_n when the strip cache S_i is evicted from the cache.

2.5 Combination of ASP and MSP

ASP suffers parallelism loss when there are only a small number of sequential reads. In other words, disks are serialized for a single stream because each request of ASP is dedicated to only one disk. However, the combination of ASP and massive stripe prefetching (MSP), which was briefly described in Section 1.2.4, resolves the parallelism loss of ASP without interfering with the ASP operation and without losing the benefit of ASP.

To combine ASP with MSP, we need the following rules: The reading by MSP does not change any of the cost variables of ASP. When ASP tries to reference a block prefetched by MSP, ASP updates C_{sp} and C_n as if the block has not been prefetched. MSP excludes blocks which are already in the cache or being read or prefetched by ASP or MSP from blocks that are assigned to be prefetched by MSP. ASP culls strip caches prefetched by MSP as well as strip prefetching.

3 Performance Evaluation

3.1 Experimental Setup

We implemented the functions of ASP and MSP in Linux kernel 2.6.18 x86_64 and added them into the RAID driver introduced in our previous work [3], which shows that the RAID driver outperforms the software-based RAID of Linux (MultiDevice) and a hardware-based RAID. We revised our earlier version of the RAID driver for a fine granularity of cache management, and disabled the contiguity transform function of our previous work for all experiments.

The system in the experiments uses dual 3.0 GHz 64-bits Xeon processors, 1 GiB of main memory, two

Adaptec Ultra320 SCSI host bus adapters, and five ST373454LC disks, each of which has a speed of 15,000 revolutions per minute (rpm) and a 75 Gbyte capacity - GiB is the abbreviation of gibibyte, as defined by the International Electrotechnical Commission. 1 GiB refers to 2^{30} bytes, whereas 1 GB refers to 10^9 bytes. [17]. The five disks comprise a RAID-5 array with a strip size of 128 KiB. A Linux kernel (version 2.6.18) for the x86_64 architecture runs on this machine; the kernel also hosts the existing benchmark programs, the ext3 file system, the anticipatory disk scheduler, and our RAID driver, namely, the Layer of RAID Engine (LORE) driver. Apart from the page cache of Linux, the LORE driver has its own cache memory of 500 MiB just as hardware RAIDs have their own cache. The block size is set to 4 KiB.

In our experiments, we compare all possible combinations of our scheme (ASP) and the existing schemes (MSP and SEQP): namely ASP, MSP, SEQP, ASP+MSP, ASP+SEQP, MSP+SEQP, and ASP+MSP+SEQP. The method of combining ASP and MSP is described in Section 2.5. ASP and MSP operate in the LORE driver while SEQP is performed by the virtual file system (VFS) that is independent of the LORE driver. By simply turning on or off the SEQP of VFS, we can easily combine SEQP with ASP or MSP without any rules.

In all the figures of this paper, SEQPX denotes a SEQP with X kibibytes of the maximum prefetch size. For example, SEQP128 indicates that the maximum prefetch size is 128 KiB. All the experimental results are obtained from six repetitions. For some of the results that exhibit significant deviation, each standard deviation is displayed in its graph.

3.2 PCMark: Evaluation of Culling

PCMark®05 records a trace of disk activity during usage of typical applications and bypasses the file system and the operating system's cache [31]. This makes the measurement independent of the file system overhead or the current state of the operating system. Because PCMark is a program for MS-Windows, we recorded the traces of PCMark and replayed the traces three times in Linux with direct IO that makes IOs bypass the operating system's cache.

Figure 6 is the results of the general hard disk drive usage of PCMark that contains disk activities from using several common applications. In this experiment, we compare ASP, strip prefetching (SP), and no prefetching. SP exhibits the highest prefetch hit rate while no prefetching provides the highest cache hit rate. ASP outperforms both SP and no prefetching at all of the cases in Fig. 6. With an over-provisioned memory, the throughput of SP is similar with that of ASP. However, ASP is significantly superior to SP with 400 MiB or less of the RAID cache. At the bast case, ASP outperforms SP by 2 times and no prefetching by 2.2 times with 400 MiB of the RAID cache.

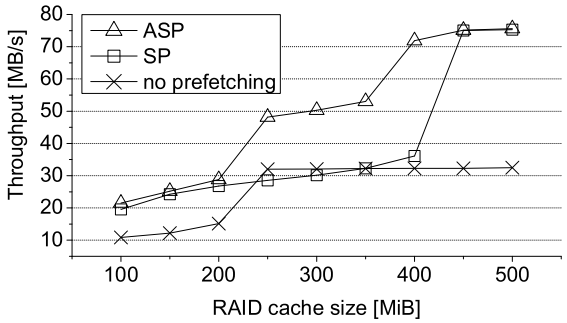


Figure 6: The experimental results of PCMark.

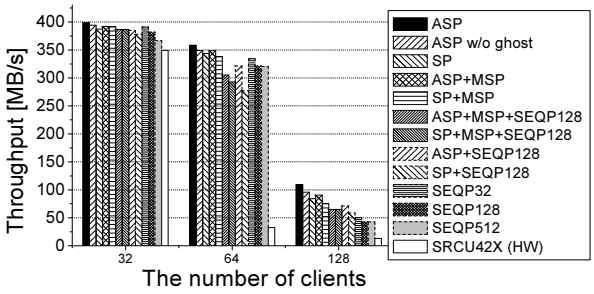


Figure 7: The experimental results of DBench (System Memory : RAID Memory = 512 MiB : 512 MiB). ASP outperforms a hardware-based RAID (Intel SRCU42X) by 11 times for 64 clients

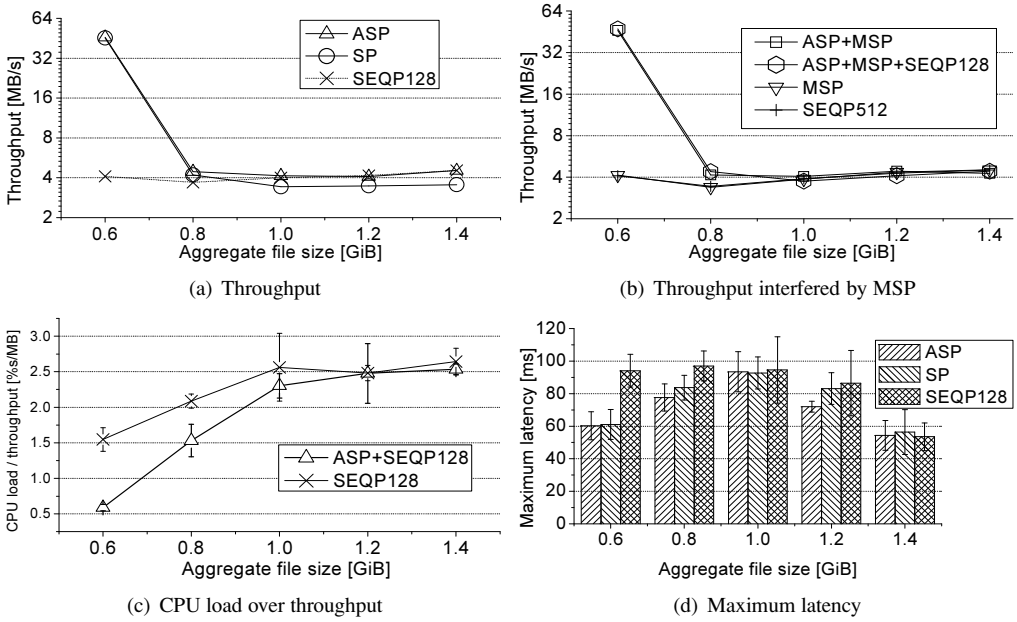
3.3 Dbench

The benchmark Dbench (version 3.04) [43] produces a local file system load. It does all the same read and write calls that the *smbd* server in Samba [37] would produce when confronted with a Netbench [30] run. Dbench generates realistic workloads consisting of so many cache hits, prefetch hits, cache misses, and writes that the ability of ASP can be sufficiently evaluated.

Figure 7 shows the results of Dbench. We divided the main memory of 1 GiB into 512 MiB for the Linux system, 500 MiB for the RAID cache, 12 MiB for the RAID driver. It took 10 minutes for each run.

Both ASP and ASP+MSP rank the highest in Fig. 7 and outperform strip prefetching (SP) by 20% for 128 clients. In this experiment, C_{sp} was always less than C_n; hence, the gain of ASP over SP originated from ACC. ASP outperforms SEQP32 by 2 times for 128 clients and a hardware-based RAID by 11 times for 64 clients and by 7.6 times for 128 clients. The hardware-based RAID, which is an Intel SRCU42X with a 512 MiB memory, has write-back with a battery pack, cached-IO, and adaptive read-ahead capabilities.

For a fair comparison, the memory of the Linux sys-



(a) Throughput

(b) Throughput interfered by MSP

(c) CPU load over throughput

(d) Maximum latency

Figure 8: Various metrics with varied aggregate file sizes. Tiobench was used to generate random reads with 4 threads and 40,000 read transactions.

tem for the hardware-based RAID was given the same size (512 MiB) as in the case of our RAID driver, and to compensate the memory occupied by the firmware of the hardware RAID, the cache of our RAID driver is tuned to 500 MiB that is slightly smaller than the internal memory of the hardware RAID. The sequential prefetching of Linux is turned on for the hardware RAID.

Sequential prefetching (SEQP) itself and combinations with SEQP significantly degrade throughput by causing independency loss. Furthermore, because redundant prefetched data that exist in both the RAID cache with ASP and the page cache of Linux with SEQP inefficiently consume cache memory. The addition of SEQP to ASP is inferior to ASP due to independency loss and redundant prefetched data. In addition, the adaptive culling operates inefficiently because prefetched data that are requested by SEQP of the host are considered cached data from the viewpoint of the RAID cache with ASP. Although ASP+SEQP128 is inferior to ASP, it outperforms SEQP128 by 67% for 128 clients.

SEQP with a large prefetch size suffers from cache wastage for a large number of processes. As shown in Fig. 7, SEQP32 outperforms SEQP512 by 4% for 64 clients and by 15% for 128 clients. In contrast, Fig. 9(a) shows that an increase in the maximum prefetch size of SEQP improves disk parallelism for a small number of processes.

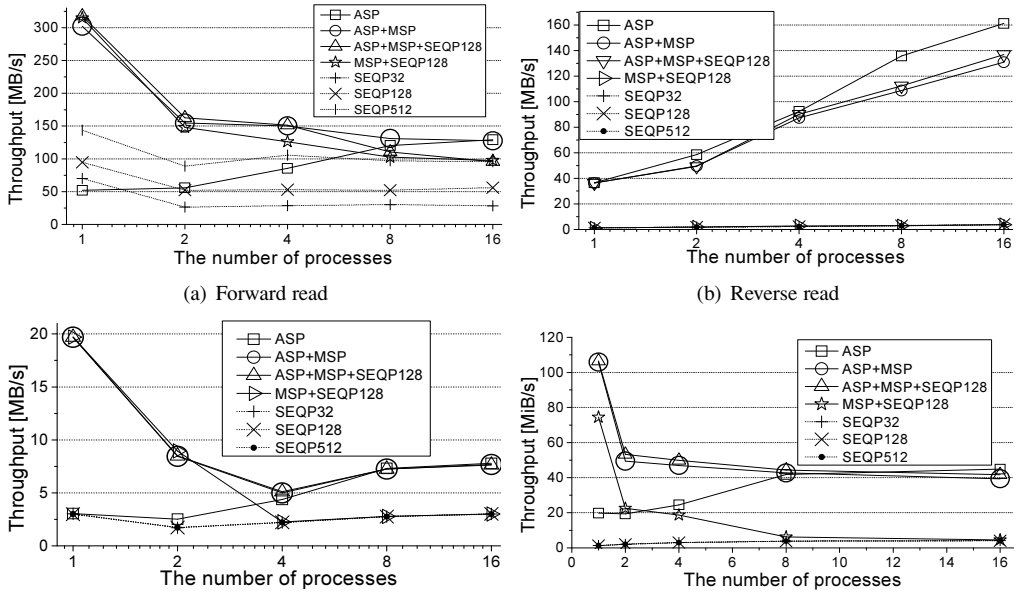
"ASP w/o ghost" in Fig. 7 indicates the improved culling effect with the past cached block. ASP outperforms ASP without the ghost by 14% for 128 clients. The addition of MSP to ASP slightly degrades ASP. However, the addition of MSP can resolve parallelism loss for small numbers of processes or a single process. Section 3.5 shows the evaluation of parallelism loss.

3.4 Tiobench: Decision Correctness

We used the benchmark Tiobench (version 0.3.3) [24] to evaluate whether the disk simulation of ASP makes the right decision on the basis of the workload property. Figure 8(a) shows the throughput of random reads with four threads, 40,000 read transactions, and 4 KiB of the block size in relation to a varied aggregate file size (workspace size). A random read with a small workspace size of 0.6 GiB exhibits such high spatial locality that ASP outperforms SEQP128. With a workspace size of 0.8 GiB, ASP outperforms both SP and SEQP. When the workspace size is equal to or larger than 1 GiB, ASP outperforms SP and show the same throughput as SEQP by deactivating SP. For random reads, SEQP does not prefetch anything. This random read with the large workspace generates so few cache hits that the adaptive cache culling does not improve the throughput of ASP.

Figure 8(b) shows that when combined with other prefetching schemes MSP is effectively disabled without interfering with the combined prefetching scheme for random reads.

Figure 8(c) shows the CPU load over throughput and the standard deviation for the random reads. Although ASP+SEQP128 requires a greater computational overhead than SEQP128, Fig. 8(c) shows that the CPU load



(a) Forward read

(b) Reverse read

(c) Stride read: the stride distance is set to 7 blocks

(d) Stride read: the stride distance is set to 3 blocks

Figure 9: Throughput obtained from the benchmark IOzone for three types of concurrent sequential reads: the forward read, the reverse read, and the stride read. The Throughput is shown in relation to the number of processes, ranging from 1 to 16, with a fixed aggregate file size of 2 GiB and a block size of 4 KiB.

over throughput of ASP+SEQP128 is less than or almost equal to that of SEQP128. Thus, the computational overhead of ASP is too low to be measured.

In spite of its throughput gain, SP may cause the latency to increase because it reads more blocks than no prefetching. However, the performance enhancement by the gain of SP decreases the latency. Figure 8(d) shows that the maximum latency of SEQP128 is longer than that of SP and ASP when SP is beneficial.

This section evaluates the decision correctness and overhead of ASP for the two types of workloads, which may or may not be beneficial to SP. However, because many types of reads exhibit spatial locality, many realistic workloads are beneficial to SP. The next sections show the performance evaluation for various workloads that exhibit spatial locality.

3.5 IOzone: Parallelism and Independency

We used the benchmark IOzone (version 3.283) [32] for the three types of concurrent sequential reads: the forward read, the reverse read, and the stride read. We vary the number of processes from one to 16 with a fixed aggregate file size of 2 GiB and a block size of 4 KiB.

As shown in Fig. 9(a), MSP boosts and dominates the forward sequential throughput when the number of processes is two or four as well as one. The combination of ASP and MSP (ASP+MSP) shows 5.8 times better throughput than ASP for a single stream because ASP suffers the parallelism loss. When the number of processes is four, ASP+MSP also outperforms ASP by 76%.

When the number of process is one, ASP+MSP outperforms SEQP512 by 134% and SEQP32 by 379%. SEQP, If sequential prefetching has the larger prefetch size, it may resolve the parallelism loss for a small number of streams. However, this approach causes cache wastage and independency loss for concurrent multiple sequential reads. When the number of streams is 16, ASP+MSP outperforms ASP+MSP+SEQP128 by 33% because a combination with SEQP gives rise to independency loss.

For the reverse sequential reads shown in Fig. 9(b), MSP and SEQP do not prefetch any block and, as a consequence, severely degrade the system throughput. ASP is the best for the reserve reads and outperforms SEQP by 41 times for 16 processes.

Figure 9(c) and 9(d) show the throughput results of the stride reads; these results were obtained with IOzone. A stride read means a sequentially discontiguous read with a fixed stride distance that is defined by the number of blocks between discontiguous reads. Stride reads cause additional revolutions in some disk models, thereby degrading the throughput. This problem was unveiled in [3]. The stride reads shown in Fig. 9 are significantly beneficial to MSP and ASP because they prefetch contiguous blocks regardless of stride requests.

Figures 9(c) and 9(d) show that as the stride distance

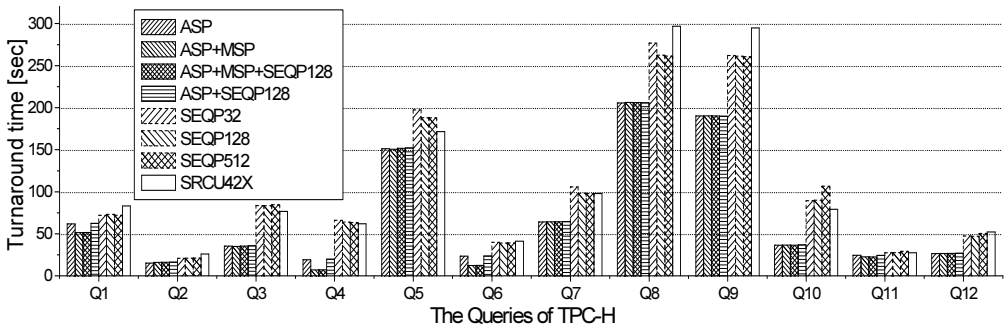


Figure 10: The turnaround time for the queries of TPC-H running on a MySQL database system.

decreases, the throughput of the combinations with ASP increases, whereas the throughput of SEQP decreases. ASP+MSP outperforms any of SEQPs by 84.6 times for a single process when the stride distance is three blocks. As the number of processes increases, MSP is deactivated and ASP dominates the throughput of the stride reads. In Fig. 9(d), the combinations with ASP outperform the combinations without ASP by at least 9.44 times when the number of processes is 16.

3.6 The TPC Benchmark™H

The TPC Benchmark™H (TPC-H) is a decision support benchmark that runs a suite of business-oriented ad-hoc queries and concurrent data modifications on a database system. Figure 10 shows the results of the queries of TPC-H running on a MySQL database system with the scale factor of one.

Most read patterns of TPC-H are stride reads and non-sequential reads with spatial locality [48], which are highly beneficial to ASP but not to SEQP. ASP and ASP+MSP outperform SEQP128 by 1.7 times and 2.2 times, respectively, on average for the 12 queries. ASP+MSP and ASP+MSP+SEQP are the best for all the 12 queries of TCP-H. Query four (Q4) delivers the best performance of ASP+MSP with 8.1 times better throughput than that of SEQP128.

The MySQL system with TPC-H exhibits neither independency loss nor enough cache hits at the global bottom. Hence, most of ASP's superior performance originates from the principle of the spatial locality of SP. In the experiments in the next section, we do not compare ASP with SP for the same reason.

3.7 Real Scenarios: Cscope, Glimpse, Link, and Booting

This section presents the results of *cscope* [38], *glimpse* [28], *link*, and *booting* as real applications.

The developer's tool *cscope*, which is used for browsing source code, can build the symbol cross-reference of C source files and allow users to browse through the C source files. Figure 11 shows the turnaround time

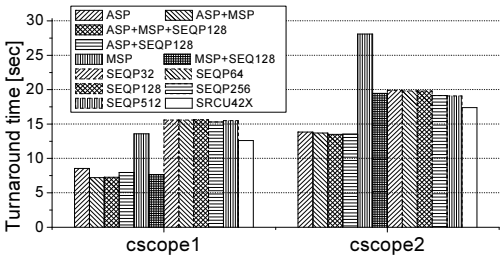


Figure 11: The turnaround time for *cscope* to build the symbol cross-reference of Linux kernel source files.

that is required for *cscope* to index the symbol cross-reference of the Linux kernel (2.6.18) source files. In Fig. 11, *cscope1* means that the symbol indexing is performed without object files that are the results of compilation. The symbol indexing of *cscope2* is performed with source files and object files.

Because *cscope* is an application that uses a single thread, SEQP does not cause independency loss. Hence, ASP+MSP+SEQP128 shows the best performance and ASP+MSP shows the next best performance. However, ASP+MSP+SEQP128 slightly outperforms ASP+MSP by 0.65% for *cscope1* and 1.2% for *cscope2*. In addition, ASP+MSP outperforms SEQP by 2.1 times in *cscope1* and by 38 % in *cscope2*.

In the *cscope2* experiment with both the source files and the object files, the performance gap between ASP+MSP and SEQP decreases. In other words, ASP prefetches unnecessary data because the required source files and the unnecessary object files are interleaved. However, ASP+MSP outperforms SEQP by at least 41%.

Figure 12 shows the execution time needed for *glimpse*, a text information retrieval application from the University of Arizona, to index the text files in "/usr/share/doc" of Fedora Core 5 Release. The results of *glimpse* resemble the results of *cscope1*. ASP+MSP and ASP+MSP+SEQP128 outperform SEQP128 by 106%.

The link of Fig. 12 shows the turnaround time for *gcc* to link the object files of the Linux kernel. The results are obtained by executing "make vmlinux" in the ker-

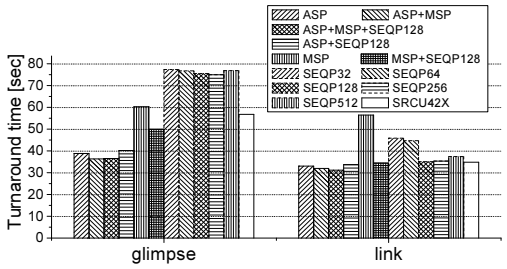


Figure 12: The turnaround time for glimpse to index the text files in "/usr/share/doc" and for *gcc* to link the object files of the Linux kernel.

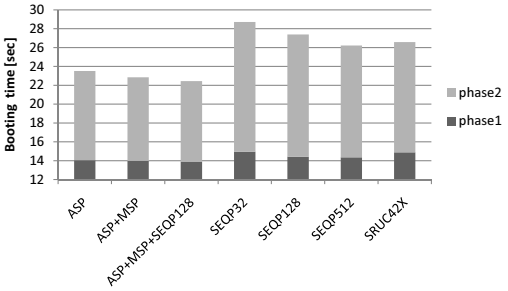


Figure 13: Linux booting speed for the run level 3: Phase 1 is the time that elapses from just before the running of the "init" process to just before the remounting of the root file system with the read-write mode. Phase 2 is the time that elapses from the end of Phase 1 to just after the completion of all the booting scripts (/etc/rc3.d/S* files).

nel source directory after executing "make vmlinux" and "rm vmlinux". The link operation requires a high computational overhead and small reads that inspect the modification date of the source files. However, ASP+MSP outperforms SEQPs by at least 10%.

Figure 13 shows the Linux booting speed. The Linux booting consists of partially random reads for scattered applications and script files. We partitioned the booting sequence into two phases. Phase 1 shown in Fig. 13 is the time that elapses from just before the running of the "init" process to just before the remounting of the root file system in read-write mode. Phase 2 is the time that elapses from the end of Phase 1 to just after the completion of all the booting scripts (/etc/rc3.d/S* files).

Phase 1 consists of CPU-bounded operations rather than I/O-bounded operations. Hence, the gain of ASP is small. Phase 2, on the other hand, requires I/O bounded operations. Because Linux booting does not produce multiple concurrent reads and consequently causes no independency loss, the best scheme for the Linux booting is ASP+MSP+SEQP128 rather than ASP+MSP. However, the performance gap is negligible. In Phase 2, ASP+MSP outperforms SEQP128 by 46% and SEQP512 by 34%.

4 Conclusion

We introduced five prefetching problems for striped disk arrays in Section 1.1. The five problems are resolved by our scheme as follows: ASP (1) resolves independency loss by aligning the read request in strips that are not laid across disks, (2) resolves parallelism loss by combining with our earlier MSP scheme, which uses the stripe size as the prefetch size to get parallelism only for small numbers of concurrent sequential reads, (3) is beneficial for non-sequential reads as well as sequential reads by exploiting the principle of spatial locality, (4) resolves the inefficient memory utilization of strip prefetching through differential feedback that maximizes the total hit rate in a given cache management scheme, (5) and using an online disk simulation, guarantees less I/O service time than no prefetching.

From the results of Dbench and IOzone, we see that SEQP suffers both parallelism loss and independent loss, but ASP and ASP+MSP are free from them. Additionally, the results of Dbench show that ASP efficiently manages prefetched data. As a result, the sum of the prefetch hit rate and cache hit rate is equal to or greater than that of strip prefetching and no prefetching. The experiments using Tiobench show that ASP has a low overhead and wisely deactivates SP if SP is not beneficial to the current workload. In the results of Dbench, ASP outperforms SEQP128 by 2.3 times and a hardware RAID controller (SRCU42X) by 11 times. The experimental result with PCMark shows that ASP is 2 times faster than SP due to the culling scheme. From the experiments using TPC-H, cscope, link, glimpse, and Linux booting, we can perceive that many realistic workloads exhibit high spatial locality. ASP+MSP is 8.1 times faster than SEQP128 for the query four of TPC-H, and outperforms SEQP by 2.2 times on average for the 12 queries of TPC-H.

Among all the prefetching schemes and combinations presented in this paper, ASP and ASP+MSP rank the highest. We implemented a RAID driver with our schemes in a Linux kernel. Our schemes have a low overhead, and can be used in various RAID systems ranging from entry-level to enterprise-class.

.....✢.....

The source code of our RAID driver is downloadable from http://core.kaist.ac.kr/dn/lore.dist.tgz, but you may violate some patent rights belonging to KAIST if you commercially use the code.

References

[1] BAEK, S. H., KIM, B. W., JOUNG, E. J., AND PARK, C. W. Reliability and performance of hierarchical RAID with multiple controllers. In *Proc. of the 20th ACM Symposium on Principles of Distributed Computing* (Aug. 2001), pp. 246–254.

[2] BAEK, S. H., AND PARK, K. H. Massive stripe cache and prefetching for massive file I/O. In *Proc. of IEEE Int'l Conf. on Consumer Electronics* (Jan. 2007), pp. 5.3–5.

[3] BAEK, S. H., AND PARK, K. H. Maxtrix-stripe-cache-based contiguity transform for fragmented writes in RAID-5. *IEEE Trans. on Computers 56*, 8 (Aug. 2007), 1040–1054.

[4] BHATTACHARYA, S., TRAN, J., SULLIVAN, M., AND MASON, C. Linux AIO performance and robustness for enterprise workloads. In *Proc. of Linux Symposium* (July 2004), pp. 63–78.

[5] BLAUM, M., BRADY, J., AND MENON, J. EVENODD: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Trans. on Computers 44*, 2 (Feb. 1995), 192–201.

[6] CHANG, F., AND GIBSON, G. A. Automatic I/O hint generation through speculative execution. In *Proc. of the 3rd Symposium on Operating Systems and Design and Implementation* (Feb. 1999), pp. 1–14.

[7] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys 26*, 2 (June 1994), 145–185.

[8] DING, X., JIANG, S., AND CHEN, F. A buffer cache management scheme exploiting both temporal and spatial localities. *ACM Trans. on Storage 3*, 2 (June 2007).

[9] GILL, B. S. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *Proc. of the 6th USENIX Conference on File and Storage Technologies* (Feb. 2008), pp. 49–65.

[10] GILL, B. S., AND BATHEN, L. A. D. AMP: Adaptive multi-stream prefetching in a shared cache. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies* (2007).

[11] GILL, B. S., AND MODHA, D. S. SARC: Sequential prefetching in adaptive replacement cache. In *Proc. of USENIX Annual Technical Conference* (Dec. 2005), pp. 293–308.

[12] GILL, B. S., AND MODHA, D. S. WOW: Wise ordering for wrties = combining spatial and temporal locality in non-volatile caches. In *Proc. of USENIX Conf. File and Storage Technologies* (2005), pp. 129–142.

[13] GRIFFIOEN, J., AND APPLETON., R. Reducing file system latency using a predictive approach. In *Proc. of USENIX Summer Technical Conf.* (June 1994), pp. 197–208.

[14] GRIMSRUD, K. S., ARCHIBALD, J. K., AND NELSON, B. E. Multiple prefetch adaptive disk caching. *IEEE Trans. on Knowledge and Data Engineering 5*, 1 (Feb. 1993), 88–103.

[15] HUANG, X.-M., LIN, C.-R., AND CHEN, M.-S. Design and performance study of rate staggering storage for scalable video in a disk-array-based video server. *IEEE Trans. on Consumer Electronics 50*, 4 (Nov. 2004), 1119–1129.

[16] HWANG, K., JIN, H., AND HO, R. S. Orthogonal striping and mirroring in distributed RAID for I/O-centric cluster computing. *IEEE Trans. on Parallel and Distributed Systems 13*, 1 (Jan. 2002), 26–44.

[17] IEC. Prefixes for binary multiples.

[18] JOSEPH, D., AND GRUNWALD, D. Prefetching using markov predictors. *IEEE Trans. on Computers 48*, 2 (Feb. 1999), 121–133.

[19] 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.

[20] KENCHAMMANA-HOSEKOTE, D., HE, D., AND HAFNER, J. L. REO: A generic RAID engine and optimizer. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies* (2007).

[21] KIM, S. H., ZHU, H., AND ZIMMERMANN, R. Zoned-RAID. *ACM Trans. on Storage 3*, 1 (Mar. 2007).

[22] KIMBREL, T., TOMKINS, A., PATTERSON, R., BERSHAD, B., CAO, P., FELTEN, E., GIBSON, G., KARLIN, A., AND LI, K. A trace-driven comparison of algroithms for parallel prefetching and caching. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation* (Oct. 1996).

[23] KROEGER, T. M., AND LONG, D. D. E. Design and implementation of a predictive file prefetching algorithm. In *Proc. of the 2001 USENIX Annual Technical Conference* (June 2001), pp. 105–118.

[24] KUOPPALA, M. Threaded I/O bench for Linux, 2002.

[25] LEI, H., AND DUCHAMP, D. An analytical approach to file prefetching. In *Proc. of the 1997 USENIX Annual Technical Conference* (Jan. 1997).

[26] 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.

[27] LIM, S.-H., JEONG, Y.-W., AND PARK, K. H. Interactive media server with media synchronized raid storage system (nossdav). In *Proc of Int'l Workshop on Network and Operating System Support for Digital Audio Video* (June 2005), pp. 177–182.

[28] MANBER, U., AND WU, S. GLIMPSE: A tool to search through entire file systems. In *Proc. of USENIX Winter 1994 Technical Conference* (San Fransisco, CA, USA, 1994), pp. 23–32.

[29] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proc. of USENIX Conf. on File and Storage Technologies* (Mar. 2003), pp. 115–130.

[30] MEMIK, G., MANGIONE-SMITH, W. H., AND HU, W. NetBench: A benchmarking suite for network processors. In *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD'01)* (2001), pp. 39–43.

[31] NIEMELÄ, S. *PCMark®05 PC Performance Analysis (white paper).* Futuremark Corporation, June 2005.

[32] NORCUTT, W. The IOzone filesystem benchmark, 2007.

[33] PALMER, M., AND ZDONIK, S. B. Fido: A cache that learns to fetch. In *Proc. of the 17th International Conf. on Very Large Data Bases* (Sept. 1991), pp. 255–264.

[34] PARK, C.-I. Efficient placement of parity and data to tolerate two disk failures in disk array systems. *IEEE Trans. on Parallel and Distributed Systems 6*, 11 (Nov. 1995), 1177–1184.

[35] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proc. of the 15th Symp. on Operating System Principles* (Dec. 1995), pp. 79–95.

[36] SCHINDLER, J., SCHOLSSER, S. W., SHAO, M., AILAMAKI, A., AND GANGER, G. R. Atropos: A disk array volume manager for orchestrated use of disks. In *Proc. of the 6th USENIX Conference on File and Storage Technologies* (Mar. 2004), pp. 159–172.

[37] SHARPE, R. Just what is SMB?, 2002.

[38] STEFFEN, J. L. Interactive examination of a C program with cscope. In *Proc. of USENIX Winter 1985 Technical Conference* (1985), pp. 170–175.

[39] THE RAID ADVISORY BOARD. *The RAIDBook: A Source Book for RAID Technology sixth edition*. Lino Lakes MN, 1999.

[40] THOMASIAN, A. Multilevel RAID disk arrays. In *Proc. of the 23rd IEEE/14th NASA Goddard Conf. on Mass Storage Systems and Technologies* (May 2006).

[41] TIAN, L., FENG, D., JIANG, H., ZHOU, K., ZENG, L., CHEN, J., WANG, Z., AND SONG, Z. PROC: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies* (2007).

[42] TOMKINS, A., PATTERSON, R. H., AND GIBSON, G. Informed multiprocess prefetching and caching. In *Proc. of the 1997 ACM Int'l Conf. on Measurement and Modeling of Computer Systems* (June 1997), pp. 100–114.

[43] VIEIRA, M., AND MADEIRA, H. A dependability benchmark for OLTP application environments. In *Proc. of the 29th Int'l. Conf. on Very Large Data Bases* (2003).

[44] VITTER, J. S., AND KRISHNAN, P. Optimal prefetching via data compression. *Joural of the ACM 43*, 5 (Sept. 1996), 771–793.

[45] WEDDLE, C., OLDHAM, M., QIAN, J., WANG, A.-I. A., AND KUENNING, G. PARAID: A gear-shifting power-aware RAID. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies* (2007).

[46] WIKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Trans. on Computer Systems 14*, 1 (Feb. 1996), 108–136.

[47] ZHANG, G., SHU, J., XUE, W., AND ZHENG, W. SLAS: An efficient approach to scaling round-robin striped volumes. *ACM Trans. on Storage 3*, 1 (Mar. 2007).

[48] ZHU, Y., AND JIANG, H. RACE: A robust adaptive caching strategy for buffer cache. *IEEE Tran. Computers 57*, 1 (Jan. 2008), 25–40.